Mastering Django: Core

精通 Django

Django 1.8 LTS 全解

Nigel George 著

安道 译

精通 Django

Django 1.8 LTS 全解

Nigel George 著

安道 译

rev 0.0.1, 2017-04-28T12:27:00+08:00

目录

致谢

首先感谢 *The Django Book* 的两位作者，Adrian Holovaty 和 Jacob Kaplan-Moss。他们为这本书打下了坚实的基础，使这一版写起来十分畅顺。

同样要感谢 Django 社区。这个社区充满生机、互助互利，很多年前发现这个“Web 框架新星”时让我这个愤世嫉俗的老商人眼前一亮。正是有社区的支持，Django 才会变得这么棒。谢谢你们！

关于作者



Nigel George 是一名商务系统开发者，专门从事使用开源技术解决常见的商务问题。他的软件开发经验丰富，为小型企业编写过数据库应用，也为澳大利亚纽卡斯尔大学的一个分布式传感器网络开发过后端和 UI。

Nigel 还有 15 年的商务技术写作经验。他为企业和澳大利亚政府部门写过很多培训手册和上百份技术规程。他从 0.96 版就开始使用 Django 了，使用 C、C#、C++、VB、VBA、HTML、JavaScript、Python 和 PHP 编写过应用。

他还写过一本关于 Django 的书——*Beginning Django CMS*，由 Apress 于 2015 年 12 月出版。

Nigel 生活在澳大利亚新南威尔士州纽卡斯尔。

导言

30 年前（截至 2014 年），我第一次在学校的第一台 Apple II 电脑中插入 5.25 英寸 DOS 3.3 磁盘，开始研究 BASIC。

那几年我使用多门语言编写了很多代码，多到让人难以置信。现在我每周还会写代码，但是所用的语言和写出的代码行数都减少了。

这些年，我见过糟糕的代码，也见过优秀的代码。就我自己而言，写出的代码也是优劣各半。奇怪的是，在我的职业生涯中，我还没真正做过程序员。我自己运营一家 IT 公司五年了，为大大小小的公司做过事，但是大都负责研发、技术和运维，从未专门做过程序员。

我一直被人使唤，去完成各项工作。

说得简单一点，生意不就是完成各项工作吗！经年累月，人们已经疲惫，不再争论弯引号和哪门语言最适合开发应用。

每学一门语言，我都会阅读大量相关的书籍，因此我知道你阅读导言是想找什么，那我们就直接进入正题吧。

为什么要关注 Django?

虽然 Django 不是完成工作唯一的 Web 框架，但是我可以确信一点，如果你想编写简洁明了的代码，想快速构建高性能、外观精美的现代网站，那么你一定能从本书中受益。

我故意不与其他语言和框架作比较，因为那无关紧要，所有语言以及使用它们构建的框架和工具都有各自的优缺点，然而根据我这些年的经验，我完全认同 Django 是出色的，能排在前列，因为使用它能快速写出安全牢固且没有缺陷的代码。

Django 能出色地完成某些任务，如果需要高级功能，它在表皮之下也提供了支持。

此外，Django 是使用 Python 构建的，这是一门被认为是最简单易学的编程语言。当然，这些优势也带来了挑战。Python 和 Django 都在表皮之下隐藏了众多强大的功能，初学者可能难以理解。本书就是为此而写的。本书旨在教你如何快速改进自己的 Django 项目，最终学会正确设计、开发和部署网站所要掌握的全部知识。

最初的 *The Django Book* 是 Adrian 和 Jacob 写的，他们相信 Django 能让 Web 开发变得更好。*The Django Book* 出版后，Django 持续发展、迅速增长，我想正是应验了这一点。与原来那本书一样，本书也是开源的，欢迎任何人做改进。你可以在本书的网站中提交评论和建议，或者给我发电子邮件，地址是 nigel@masteringdjango.com。我与很多人一样，在使用 Django 的过程中身心愉悦。Django 与 Adrian 和 Jacob 期望的一样，让人欢心，是开发的好帮手。

关于本书

这是一本讲解 Django 的书。Django 是一个 Web 开发框架，能节省 Web 开发的时间，让整个过程充满欢乐。使用 Django 开发 Web 应用能达到事半功倍的效果。本书对 *The Django Book* 做了全面的修订和升级。*The Django Book* 最初由 Apress 于 2007 年出版，题为 *The Definitive Guide to Django: Web Development Done*

Right，后来又由两位作者在 2009 年重新出版，而且书名换成了 *The Django Book*。后者是一个开源项目，基于 GUN 自由文档许可证（GFDL）发布。

本书可以认为是 *The Django Book* 的非官方第三版。不过，我是否有这个荣幸，还要看 Jacob 和 Django 社区是否认可。对我个人而言，我十分希望 *The Django Book* 能够得到更新，因为我就是从那本书入门的。为了保留 Adrian 和 Jacob 对 *The Django Book* 的最初期许，本书的源码在本书的网站上也可以免费获取。

本书的主要目的是把你打造成 Django 专家。本书集中讲解两方面的内容。首先，深入说明 Django 的机制，教你使用它构建 Web 应用。其次，适当讨论高级概念，说明如何在项目中有效使用相关的工具。阅读本书你将学会快速开发强大网站所需的技能，而且写出的代码简洁、易于维护。

本书的第二个目的（没那么重要）是为程序员提供一份关于 Django 长期支持（Long Term Support，LTS）版本的手册。目前，Django 已经成熟，很多重要的商业网站都使用它开发。因此，本书意欲成为采用 Django 1.8 LTS 的商业网站的最新权威参考资源。本书电子版会一直更新，直到对 Django 1.8 的支持结束（2018 年）。

如何阅读本书

写作本书时，我尽量参照原书，在可读性和作为参考手册两方面做了平衡。然而，2007 年之后，Django 有了长足的发展，功能和灵活性都提高了，从而引入了额外的复杂度。在一众 Web 应用框架中，Django 仍是学习曲线最短的，但是若想掌握 Django，还是要学习很多知识。本书保留了原书的“通过实例学习”策略，不过某些较复杂的章节（如数据库配置）往后移了。因此，我们将先使用默认的配置学习 Django 的机理，掌握一定的知识后，再讨论更高级的话题。

鉴于此，我建议你按顺序从第 1 章读到第 13 章。这几章是使用 Django 的基础，读完之后便能构建和部署 Django 驱动的网站。具体而言，第 1-6 章是“基础课程”，第 7-12 章是较高级的话题，第 13 章则涵盖部署。余下的几章（14-21）专注于特定的 Django 功能，可以按任何顺序阅读。附录是参考手册。你可能会经常翻阅那些附录和 Django Project 网站中的免费文档，回顾句法或者快速概览 Django 某个功能的作用。

所需的编程知识

本书的读者应该知道过程式编程和面向对象编程的基础知识：控制结构（如 `if`、`while`、`for`）、数据结构（列表、散列/字典）、变量、类和对象。如你所想，有 Web 开发经验更好，但这不是必须的。本书会尽量说明 Web 开发的最佳实践。

所需的 Python 知识

说到底，Django 只是一系列使用 Python 编程语言编写的库。使用 Django 开发网站就是使用这些库编写 Python 代码。因此，学习 Django 是学习如何使用 Python 编程，以及理解 Django 库的机理。如果你有使用 Python 编程的经验，阅读本书就没有障碍。总的来说，Django 代码没有多少“魔法”（即难以说明和理解的编程技巧）。对你来说，学习 Django 就是学习 Django 的约定和 API。

即便没有 Python 编程经验，你也会喜欢上它的。Django 易于学习，用起来得心应手。本书虽然不含完整的 Python 教程，但是会适时强调重要的 Python 特性和功能，尤其是无法立即理解的代码。不过，我还是建议你阅读官方的 Python 教程。我还建议你阅读 Mark Pilgrim 写的 *Dive Into Python*，这是一本免费书，可以在线阅读，地址是 http://www.diveintopython.net/；这本书也由 Apress 出版过纸质版。

所需的 Django 版本

本书涵盖 Django 1.8 Long Term Support（LTS）。这是 Django 的长期支持版本，在 2018 年 4 月之前会一直

得到全面支持。

如果你用的是早期的 Django 版本，建议你升级到最新的 Django 1.8 LTS。本书印刷时（2016 年 7 月），当前最新的 Django 1.8 LTS 生产版本是 1.8.13。

如果你使用的是 Django 的后续版本，要注意，虽然 Django 的开发者会尽量维护向后兼容性，但是偶尔会引入不向后兼容的改动。发布记中有每一次发布的改动，地址是 https://docs.djangoproject.com/en/dev/releases/。

寻求帮助

Django 的一大优势是，它的用户社区友善、乐于助人。如果你对 Django 的任何方面有疑问，从安装到应用设计，到数据库设计和部署，都可以在线询问。

- 数千名 Django 用户经常逛 django-users 邮件列表，在里面回答问题。免费注册地址 http://www.djangoproject.com/r/django-users。
- Django 用户在 Django IRC 频道中实时聊天和互帮互助。加入 Freenode IRC 网络中的 `#django` 频道吧！

Django 简介

优秀的开源软件不断涌现，因为总是有那么一些聪明的开发者遇到一些问题无法使用现有的方案有效解决。Django 就是一例。Adrian 和 Jacob 早已退出这个项目，但是它们为 Django 定的基调始终没变。正是这种基于实战的基础使得 Django 如此成功。鉴于他们所做的巨大贡献，我觉得最好由他们来介绍 Django（根据原书做了编辑和重排）。

介绍 Django

Adrian Holovaty 和 Jacob Kaplan-Moss
2009 年 12 月

在 Web 早期阶段，开发者手动编写每个页面。更新网站要编辑 HTML；重新设计要重新制作每一个网页，而且一次只能改一个网页。随着网站体量的增大，这种方式立马变得繁琐、浪费时间，最终变得不切实际。

NCSA（National Center for Supercomputing Applications，国家超级计算应用中心，第一款图形 Web 浏览器 Mosaic 就是在这里开发出来的）一群富于创新的黑客解决了这个问题，他们让 Web 服务器派生外部程序，动态生成 HTML。他们把这一协议称为*通用网关接口*（Common Gateway Interface，CGI），自此，Web 完全变了样。如今，很难想象 CGI 带来的变革：CGI 不再把 HTML 页面视作硬盘中存储的文件，而是把页面看做资源，可以按需动态生成。

CGI 的开发促使了第一代动态网站的出现。然而，CGI 自身也有问题：CGI 脚本包含大量重复的样板代码，导致代码难以复用，而且新手难以编写和理解。

PHP 解决了这些问题中的多数，在 Web 开发界引起了一阵风暴。PHP 现在是创建动态网站最流行的工具，多门类似的语言（ASP、JSP，等等）都参照了 PHP 的设计原则。PHP 的主要创新是易于使用：PHP 代码直接嵌入普通的 HTML 中；对学过 HTML 的人来说，学习曲线极为平缓。

但是，PHP 也有自身的问题：就是因为易于使用，写出的代码凌乱、重复，设计不周。更糟的是，PHP 没有为程序员提供多少防止安全漏洞的保护机制，很多 PHP 开发者意识到这一点再去学习相关的知识就晚了。

上述问题以及类似的缺陷直接促使了“第三代”Web 开发框架的涌现。Web 开发的新方式也提升了人们的雄心，现在 Web 开发者每天所做的工作越来越多。

Django 就是为了迎接这些雄心而诞生的。

Django 的历史

Django 是从真实的应用中成长起来的，由美国堪萨斯州劳伦斯的一个 Web 开发团队编写。它诞生于 2003 年秋天，那时 *Lawrence Journal-World* 报社的 Web 开发者 Adrian Holovaty 和 Simon Willison 在尝试使用 Python 构建应用。

World Online 团队负责制作和维护本地的几个新闻网站，在新闻界特有的快节奏开发环境中逐渐发展壮大。那些网站（包括 LJWorld.com、Lawrence.com 和 KUsports.com）的记者（和管理层）不断要求增加功能，而且整个应用要在紧张的周期内快速开发出来，通常只有几天或几小时。因此，Simon 和 Adrian 别无他法，只能开发一个节省时间的 Web 开发框架，这样他们才能在极短的截止日期之前构建出易于维护的应用。

经过一段时间的开发后，那个框架已经足够驱动世界上最大的在线网站了。2005 年夏天，团队（彼时 Jacob Kaplan-Moss 已经加入）决定把框架作为开源软件发布出来。他们在 2005 年 7 月发布了那个框架，将其命名为 Django——取自爵士吉他手 Django Reinhardt。

这段历史相当重要，因为说清了两件要事。首先是 Django 的“发力点”。Django 诞生于新闻界，因此它提供了几个特别适合“内容型”网站使用的功能（如管理后台，参见第 5 章）。这些功能适合 Amazon.com、craigslist.org 和 washingtonpost.com 这样动态的数据库驱动型网站使用。

不过，不要因此而灰心。虽然 Django 特别适合开发这种网站，但是这并没有阻碍它成为开发任何动态网站的有效工具。（某些方面“特别”高效与某些方面不高效是由区别的。）

第二点是，Django 最初的理念塑造了开源社区的文化。Django 是从真实代码中提取出来的，而不是科研项目或商业产品，它专注于解决 Django 的开发者自身所面对的问题。因此，Django 一直在积极改进，几乎每一天都有变化。Django 框架的维护者一心确保它能节省开发者的时间，确保开发出的应用易于维护，而且在高负载下的性能良好。

使用 Django 能在极短的时间内构建全面动态的网站。Django 的主旨是让你集中精力在有趣的工作上，减轻重复劳作的痛苦。为此，它为常用的 Web 开发模式提供了高层抽象，为常见的编程任务提供了捷径，还为解决问题提供了清晰的约定。与此同时，Django 尽量做到不挡路，允许你在必要时脱离框架。

我们之所以写这本书，是因为我们坚信，Django 能把 Web 开发变得更好。本书旨在教你如何快速改进自己的 Django 项目，最终学会正确设计、开发和部署网站所要掌握的全部知识。

第 1 章 新手入门

开始使用 Django 之前有两件重要的事要做：

1. 安装 Django（明摆着的）
2. 适当理解模型-视图-控制器（Model-View-Controller，MVC）设计模式

首先要安装 Django，这一步特别简单，本章前半部分会详细说明。第二点同样重要，如果你刚接触编程，或者之前使用的编程语言没有把数据和显示数据的逻辑区分开，更要理解。Django 的哲学建立在“松耦合”之上，这正是 MVC 背后的哲学。本书会不断深入说明松耦合和 MVC，如果你对 MVC 知之甚少，最好别跳过本章后半部分，因为理解 MVC 之后，理解 Django 就容易多了。

1.1 安装 Django

学习使用 Django 之前，必须在电脑中安装一些软件。幸好，这个过程很简单，分为下述三步：

1. 安装 Python
2. 安装 Python 虚拟环境
3. 安装 Django

如果你不知道怎么做，别担心，本章假定你以前从未在命令行中安装过软件，会一步步说明安装过程。

本节针对使用 Windows 系统的读者。虽然很多 Django 用户使用 *nix 和 OS X，但是大多数新手用的是 Windows。如果你用的是 Mac 或 Linux，网上有大量资源——首先应该阅读 Django 的安装说明。

对 Windows 用户来说，你的电脑应该可以运行任何最近的 Windows 版本（Vista、7、8.1 或 10）。本章还假设你在桌面电脑或笔记本电脑中安装 Django，而且使用开发服务器和 SQLite 运行书中的所有示例代码。目前，对新手来说，这是安装 Django 最简单、最好的方式。

如果你想使用更为高级的方式安装 Django，请阅读第 13 章、第 20 章和第 21 章。

提示

如果使用 Windows，我建议你尝试使用 Visual Studio 做 Django 开发。为了给 Python 和 Django 程序员提供支持，Microsoft 投入了很多精力，为 Python/Django 提供了全面的 IntelliSense 支持，而且把 Django 的所有命令行工具都集成到 VS IDE 中了。

最重要的一点是，VS 完全免费。我知道没人会想到 M$ 会免费提供，但这是千真万确的！

附录 G 详细说明了 Visual Studio Community 2015 的安装方法，还为在 Windows 中做 Django 开发给出了几个小贴士。

1.2 安装 Python

Django 完全使用 Python 编写，因此安装框架的第一步是安装 Python。

1.2.1 Python 版本

Django 1.8 LTS 支持 Python 2.7、3.3、3.4 和 3.5。对各个 Python 版本来说，Django 只支持最新的微版本（A.B.C）。

如果你只想试用 Django，使用 Python 2 还是 Python 3 没有关系。然而，如果你准备开发一个线上网站，应该把 Python 3 作为第一选择。Python 的维基使用简洁的语言说明了这么做的原因：

> 简单来说，Python 2.x 已经过时，Python 3.x 是这门语言的现在和未来。

除非有特别理由使用 Python 2（例如，要使用过时的库），否则应该使用 Python 3。

> **注意**
>
> 本书所有代码示例都是用 Python 3 编写的。

1.2.2 安装过程

如果你用的是 Linux 或 Mac OS X，系统可能已经预装了 Python。在命令提示符（在 OS X 中使用 Applications/Utilities/Terminal）输入 `python`，如果看到类似下面的输出，说明已经安装了 Python：

```
Python 2.7.5 (default, June 27 2015, 13:20:20)
[GCC x.x.x] on xxx Type "help", "copyright", "credits"
or "license" for more information.
>>>
```

> **提醒**
>
> 可以看出，上述示例中的 Python 交互模式在 Python 2.7 中运行。没有经验的用户可能受骗。在 Linux 和 Mac OS X 设备中，经常会同时安装 Python 2 和 Python 3。如果你的系统是这样，要在所有命令前面输入 `python3`，而不是 `python`，这样才会使用 Python 3 运行 Django。

假如你的系统中没有安装 Python，那么首先要下载安装程序。访问 https://www.python.org/downloads/，点击文字为“Download Python 3.x.x”的黄色大按钮。

写作本书时，Python 的最新版是 3.5.1。你阅读本书时，可能有更新，所以版本号可能稍有不同。

别下载 2.7.x 版，这是旧版 Python。本书所有代码都是用 Python 3 编写的，如果尝试使用 Python 2 运行，会遇到兼容问题。

Python 安装程序下载完毕后，双击下载文件夹里的“python-3.x.x.msi”，运行安装程序。安装过程与其他 Windows 程序一样，如果你以前安装过软件，那就没有问题，不过有一个选项极为重要，一定要定制。

提醒

别忘了接下来的那一步，Windows 中的多数问题都是由于没有正确设置 pythonpath （对 Python 较为重要的一个变量）。

默认情况，Python 可执行文件不会加入 Windows 的 PATH 变量。若想正确使用 Django，PATH 变量中必须有 Python。幸好，这个问题易于解决：

- 对 Python 3.4.x 来说，安装程序打开定制窗口时，“Add python.exe to Path”选项没有选中，一定要把这个选项改为“Will be installed on local hard drive”，如图 1-1 所示。
- 对 Python 3.5.x 来说，安装之前一定要选中“Add Python 3.5 to PATH”（图 1-2）。

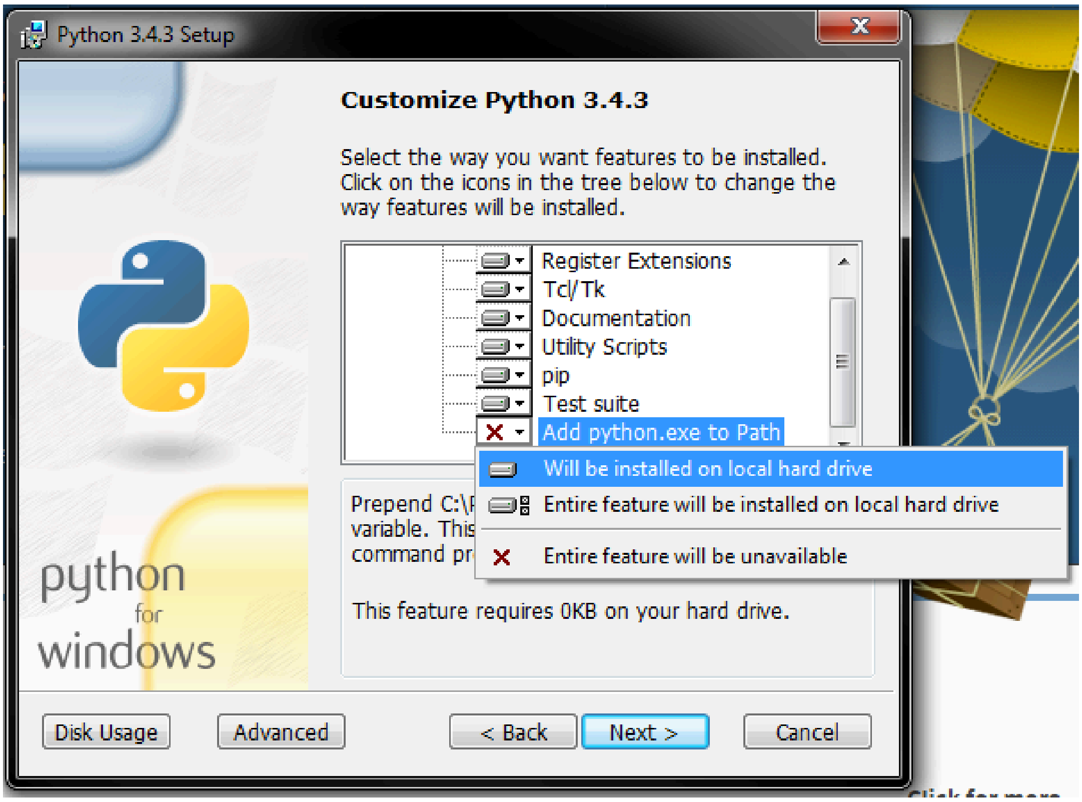


图 1-1：把 Python 添加到 PATH 中（3.4.x 版）

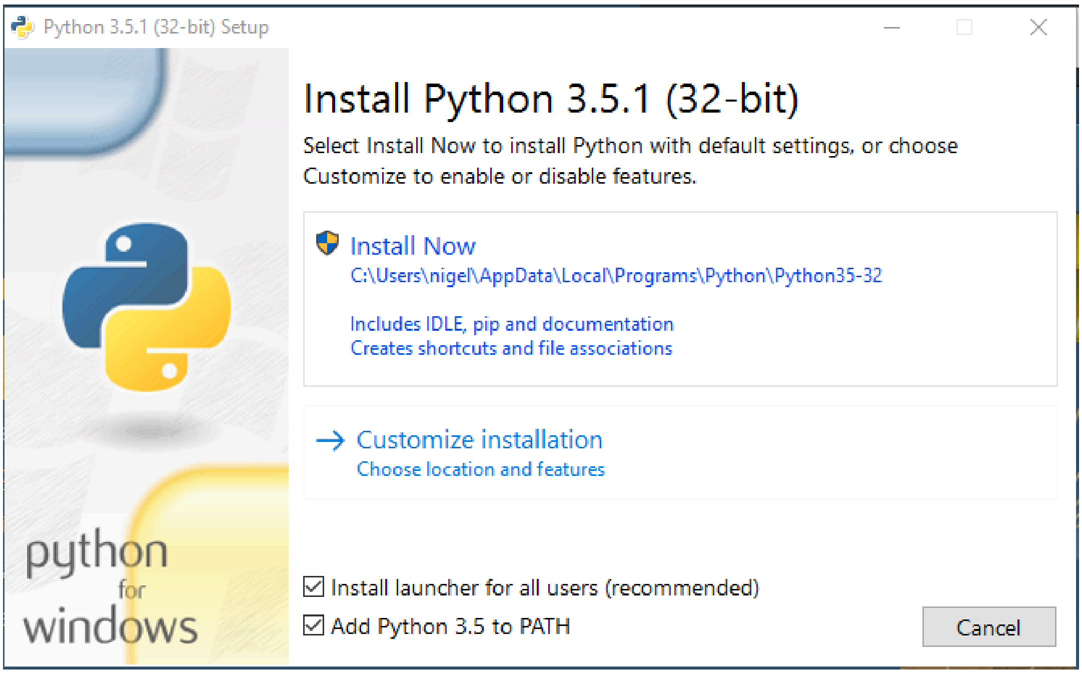


图 1-2：把 Python 添加到 PATH 中（3.5.x 版）

安装好 Python 之后，应该重新打开命令窗口，然后在命令提示符中输入 `python`，看有没有输出类似下面的内容：

```
Python 3.5.1 (v3.5.1:37a07cee5969, Dec  6 2015,
01:38:48) [MSC v.1900 32 bit (Intel)] on win32 Type "help", "copyright", "credits"
or "license" for more information.
>>>
```

既然已经打开命令提示符，那就再做一件重要的事吧。按 CTRL-C 键，退出 Python。在命令提示符中输入下述命令，然后按回车键：

```
python -m pip install -U pip
```

这个命令的输入类似下面这样：

```
C:\Users\nigel>python -m pip install -U pip
Collecting pip
  Downloading pip-8.1.2-py2.py3-none-any.whl (1.2MB)
    100% |################################| 1.2MB 198kB/s
Installing collected packages: pip
Found existing installation: pip 7.1.2
  Uninstalling pip-7.1.2:
    Successfully uninstalled pip-7.1.2
Successfully installed pip-8.1.2
```

现在，你无须知道这个命令的具体作用。简单来说，`pip` 是 Python 的包管理工具，用于安装 Python 包。pip 是“Pip Installs Packages”的递归缩写。pip 对安装过程中接下来的一步十分重要，但是首先我们要确保运行的是 pip 的最新版（写作本书时是 8.1.2 版），上述命令就是这个作用。

1.3 安装 Python 虚拟环境

提示

如果你准备使用 Microsoft Visual Studio（VS），别再往下读了，请跳到附录 G。VS 只要求你安装 Python，剩下的事都交给 VS 的集成开发环境（Integrated Development Environment，IDE）。

电脑中的软件相互依赖，每个程序都要依赖某些其他程序，而且要找到运行其他软件的设置（环境变量）。

编写新软件程序时，可能（经常）要修改其他软件所需的依赖或环境变量。这一步可能会导致各种问题，因此要避免。

Python 虚拟环境能解决这个问题。它把软件所需的全部依赖和环境变量包装到一个文件系统中，与电脑中的其他软件隔离开。

提醒

看过其他教程的读者可能会注意到，别的教程往往说这一步是可选的。我不这么认为，很多 Django 核心开发者也不同意这么做。

使用虚拟环境开发 Python 应用程序（包括 Django）的优点很明显，在此无需赘述。对新手来说，你只要相信我就行了：在虚拟环境中做 Django 开发不是可选的。

Python 的虚拟环境工具是 `virtualenv`，可以在命令行中使用 pip 安装：

```
pip install virtualenv
```

命令窗口中的输出应该像下面这样：

```
C:\Users\nigel>pip install virtualenv
  Collecting virtualenv
  Downloading virtualenv-15.0.2-py2.py3-none-any.whl (1.8MB)
    100% |################################| 1.8MB 323kB/s
Installing collected packages: virtualenv
Successfully installed virtualenv-15.0.2
```

安装好 `virtualenv` 之后，输入下述命令，为你的项目创建一个虚拟环境：

```
virtualenv env_mysite
```

提示

网上的示例大都使用“env”做环境名称。这样不好，主要原因是，可能有多个虚拟环境测试不同的配置，而“env”不便于区分各个环境。例如，你可能会开发一个必须使用 Python 2.7 和 Python 3.4 运行的应用程序。此时，把环境命名为“env_someapp_python27”和“env_someapp_python34”，比命名为“env”和“env1”更易于区分二者。

这里，简单起见，我把环境命名为“env_mysite”，因为我们的项目只会使用这么一个虚拟环境。上述命令的输出应该类似下面这样：

```
C:\Users\nigel>virtualenv env_mysite Using base prefix 'c:\\users\\nigel\\appdata\\lo\
cal\\programs\\python\\python35-32'
New python executable in C:\Users\nigel\env_mysite\Scripts\python.exe
Installing setuptools, pip, wheel...done.
```

等 `virtualenv` 设置好新的虚拟环境之后，打开 Windows 资源管理器，看一下 `virtualenv` 为我们创建了什么。在家目录中，会看到一个名为 `\env_mysite` 的文件夹（或者为虚拟环境起的其他名称）。打开那个文件夹，会看到下述结构：

```
\Include
\Lib
\Scripts
\src
```

`virtualenv` 创建了一个完整的 Python 安装，它与其他软件是隔离开的，因此开发项目时不会影响系统中的其他软件。

若想使用这个新建的 Python 虚拟环境，要将其激活。回到命令提示符，输入下述命令：

```
env_mysite\scripts\activate
```

这个命令会运行虚拟环境中 `\scripts` 文件夹里的 `activate` 脚本。你会发现，现在命令提示符变了：

```
(env_mysite) C:\Users\nigel>
```

命令提示符开头的 `(env_mysite)` 是告诉你，你正在那个虚拟环境中。下一步安装 Django。

1.4 安装 Django

至此，我们安装了 Python，也搭建了虚拟环境。接下来安装 Django 就超级简单了，只需输入下述命令：

```
pip install django==1.8.13
```

上述命令告诉 pip，让它把 Django 安装到虚拟环境中。命令的输出应该类似下面这样：

```
(env_mysite) C:\Users\nigel>pip install django==1.8.13
Collecting django==1.8.13
  Downloading Django-1.8.13-py2.py3-none-any.whl (6.2MB)
    100% |################################| 6.2MB 107kB/s
Installing collected packages: django
Successfully installed django-1.8.13
```

这里，我们明确告诉 pip 安装 Django 1.8.13，即写作本书时 Django 1.8 LTS 的最新版。安装 Django 时，最好访问 Django Project 网站，看看 Django 1.8 LTS 的最新版是什么。

> **提示**
>
> 以防你好奇，我告诉你，输入 `pip install django` 会安装 Django 的最新稳定版。如果想知道如何安装 Django 的最新开发版，请阅读第 20 章。

安装好之后，我们应该花点时间做些测试。在虚拟环境的命令提示符中输入 `python`，然后按回车键，启动 Python 交互式解释器。如果成功安装，应该能导入 `django` 模块：

```
(env_mysite) C:\Users\nigel>python Python 3.5.1 (v3.5.1:37a07cee5969, Dec  6 2015,
```

```
01:38:48) [MSC v.1900 32 bit (Intel)] on win32 Type "help", "copyright", "credits"
or "license" for more information.
>>> import django
>>> django.get_version()
'1.8.13'
```

1.5 安装数据库

完成本书的示例不必做这一步。Django 默认自带 SQLite。这个数据库无需配置。如果你想使用“大型”数据库引擎，如 PostgreSQL、MySQL 或 Oracle，请阅读第 21 章。

1.6 新建项目

安装好 Python、Django 和数据库服务器/库（可选）之后，可以开始开发 Django 应用程序了。[1] 第一步是创建项目。

一个项目是一个 Django 实例的一系列设置。如果这是你第一次使用 Django，要做些初始设置。具体来说，你要自动生成一些代码，创建一个 Django 项目，即 Django 实例的一系列设置，包括数据库配置、Django 相关的选项和应用程序相关的设置。

我假设你现在仍然运行着前一步那个虚拟环境。如果没有，请执行 `env_mysite\scripts\activate\` 命令，再次激活那个环境。在虚拟环境的命令行中运行下述命令：

```
django-admin startproject mysite
```

上述命令会在当前目录（这里是 `\env_mysite\`）中新建 `mysite` 目录。如果你不想在根目录中创建项目，可以新建一个目录，然后进入其中，再运行 `startproject` 命令。

> **提醒**
>
> 不要使用 Python 或 Django 的组件名命名项目。具体而言，不要使用“django”（与 Django 冲突）或“test”（与 Python 内置的一个包冲突）这样的名称。

我们来看一下 `startproject` 为我们创建了什么：

```
mysite/
  manage.py
  mysite/
    __init__.py
    settings.py
    urls.py
    wsgi.py
```

这些文件是：

- 外层的 `mysite/` 根目录是项目的容器。这个目录的名称对 Django 没有什么作用，你可以根据喜好重命

1. 在中文版中，“应用程序”对应于“application”，“应用”对应于“app”。在一般的 Web 开发中，这二者几乎没什么区别，但是在 Django 中二者有一个明显的区别：application 是指一个完整的 Web 程序，而 app 是指一个可复用的包，可以“插入”其他 Django 应用程序中。望读者在阅读时注意区分。——译者注

名。

- `manage.py` 是一个命令行实用脚本，可以通过不同的方式与 Django 项目交互。这个文件的详细说明参见 Django Project 网站。
- 内部的 `mysite/` 目录是项目的 Python 包。导入这里面的内容时要使用目录的名称（如 `mysite.urls`）。
- `mysite/init.py` 是一个空文件，目的是让 Python 把这个目录识别为 Python 包。（如果你刚接触 Python，关于包的说明请阅读 Python 官方文档。）
- `mysite/settings.py` 是 Django 项目的设置/配置。附录 D 对设置做了详细说明。
- `mysite/urls.py` 是 Django 项目的 URL 声明，即 Django 驱动的网站的“目录”。第 2 章和第 7 章将进一步说明 URL。
- `mysite/wsgi.py` 是兼容 WSGI 的 Web 服务器的入口点，用于伺服项目。详情参见第 13 章。

1.6.1 Django 的设置

接下来，编辑 `mysite/settings.py`。这是一个普通的 Python 模块，在模块层定义了一些变量，表示 Django 的设置。编辑 `settings.py` 的第一步是把 `TIME_ZONE` 设为你所在的时区。注意文件顶部的 `INSTALLED_APPS` 设置，其值是这个 Django 实例中激活的全部 Django 应用。一个应用可以在多个项目中使用，而且应用可以打包，供其他项目使用。默认情况下，`INSTALLED_APPS` 包含下述应用，这些都是 Django 自带的：

- `django.contrib.admin`：管理后台
- `django.contrib.auth`：身份验证系统
- `django.contrib.contenttypes`：内容类型框架
- `django.contrib.sessions`：会话框架
- `django.contrib.messages`：消息框架
- `django.contrib.staticfiles`：管理静态文件的框架

Django 项目默认包含这些应用，这是为常见场景所做的约定。其中某些应用要使用数据库表，因此使用之前要在数据库中创建所需的表。为此，运行下述命令：

```
python manage.py migrate
```

`migrate` 命令查看 `INSTALLED_APPS` 设置，根据 `settings.py` 文件中的数据库设置，以及应用自带的数据库迁移（后文说明）创建所需的数据库表。每执行一个迁移都会看到一个消息。

1.6.2 开发服务器

下面确认 Django 项目是否能运行。进入外层 `mysite` 目录（如果你现处别的位置），然后运行下述命令：

```
python manage.py runserver
```

在命令行中将看到下述输出：

```
Performing system checks...

0 errors found June 12, 2016 - 08:48:58 Django version 1.8.13, using settings 'mysite\
.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CTRL-BREAK.
```

我们启动的是 Django 开发服务器，这是一个轻量级 Web 服务器，完全使用 Python 编写。Django 自带这个服务器，以便快速开发，而不用花时间配置生产服务器（如 Apache）——这一步在准备好部署到生产环境时再做。

注意，别在任何生产环境中使用这个服务器，它只能在开发过程中使用。

现在，开发服务器正在运行，在 Web 浏览器中访问 http://127.0.0.1:8000/。你会看到一个淡蓝色背景的“Welcome to Django”页面（图 1-3）。这证明 Django 能正常运行！

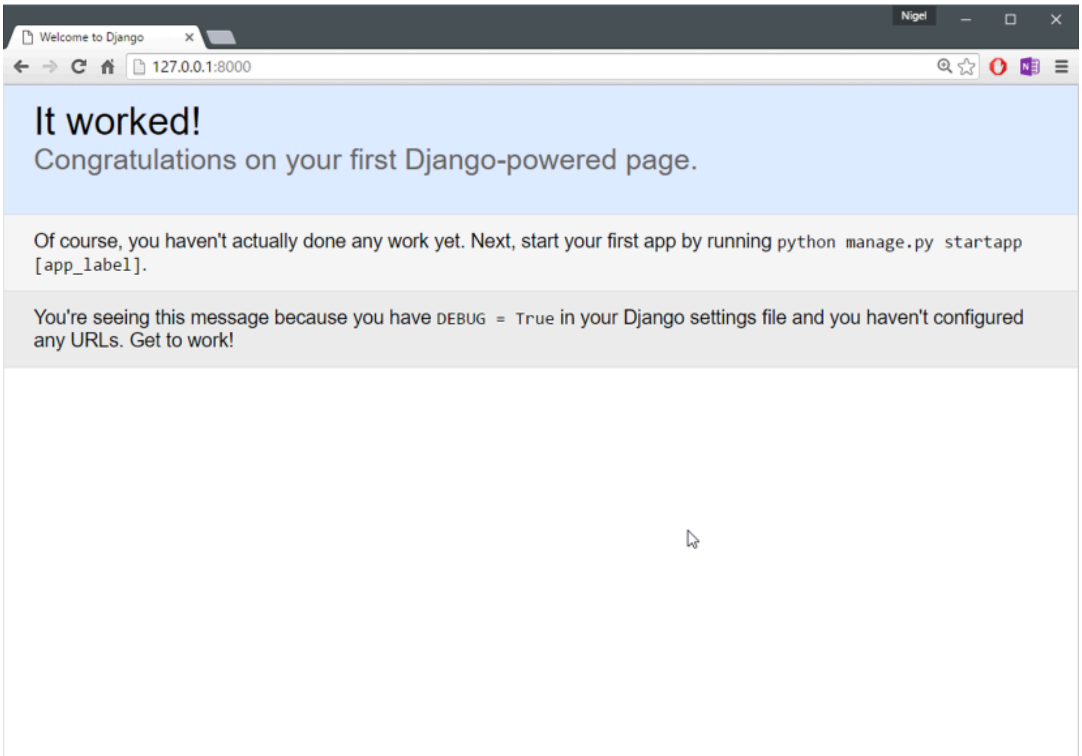


图 1-3：Django 的欢迎页面

自动重新加载服务器

开发服务器会根据需要在每次请求时自动重新加载 Python 代码，我们无需自己动手重启服务器，改动的代码自动生效。然而，有些操作，如添加文件，不会触发重启，因此在这些情况下需要自己动手重启服务器。

1.7 模型-视图-控制器设计模式

MVC 这个概念存在很长时间了，但是随着互联网的发展才被更多的人熟知，因为它是设计客户端-服务器应用的最佳方式。所有最好的 Web 框架都围绕 MVC 概念构建。虽然可能引起激烈的争论，但我还是要说，不使用 MVC 设计 Web 应用是错误的做法。就概念层次而言，MVC 设计模式非常容易理解：

- 模型（M）是数据的表述。它不是真正的数据，而是数据的接口。使用模型从数据库中获取数据时，无需知道底层数据库错综复杂的知识。模型通常还会为数据库提供一层抽象，这样同一个模型就能使用不同的数据库。
- 视图（V）是你看到的界面。它是模型的表现层。在电脑中，视图是你在浏览器中看到的 Web 应用的

页面，或者是桌面应用的 UI。视图还提供了收集用户输入的接口。

- 控制器（C）控制模型和视图之间的信息流动。它通过程序逻辑判断通过模型从数据库中获取什么信息，以及把什么信息传给视图。它还通过视图从用户那里收集信息，并且实现业务逻辑：变更视图，或者通过模型修改数据，或者二者兼具。

真正让人困惑的是如何理解各层的作用，不同的框架往往会使用不同的方式实现同样的功能。一个框架“专家”可能会说某个函数属于视图，另一个专家可能强烈反对，觉得应该放在控制器中。

对于真正做事的程序员来说，我们无需关心这个问题，因为完全没关系。只要理解 Django 实现 MVC 模式的方式，我们就能自由运用，把工作做好。不过，在讨论组中看人激烈争论是打发无聊的不错方式……

Django 严格遵守 MVC 模式，但是有自己的实现逻辑。“C”部分由框架处理，多数时候，我们的工作在模型、模板和视图中，因此 Django 经常被称为 MTV 框架。在 MTV 开发模式中：

- M 表示“模型”，即数据访问层。这一层包含所有与数据相关的功能：访问数据的方式、验证数据的方式、数据的行为、数据之间的关系。第 4 章将深入探讨 Django 的模型。
- T 表示“模板”，即表现层。这一层包含表现相关的决策：在网页或其他文档类型中如何显示某个东西。第 3 章将探讨 Django 的模板。
- V 表示“视图”，即业务逻辑层。这一层包含访问模型和选择合适模板的逻辑。你可以把视图看做模型和模板之间的桥梁。下一章将讨论 Django 的视图。

在名称的使用上，这可能是 Django 唯一的不足，因为 Django 的视图更像是 MVC 中的控制器，而 MVC 中的视图是 Django 中的模板。乍一看有点难以理解，不过作为做事的程序员来说，无需纠结这个问题。教授 Django 的人才应该深究。当然，喜欢叫板的人也是。

1.8 接下来

我们已经安装了所需的一切，而且运行了开发服务器，接下来转向 Django 的视图，学习使用 Django 伺服网页的基础知识。

第 2 章 视图和 URL 配置

前一章说明了如何创建 Django 项目，以及如何运行 Django 开发服务器。本章学习使用 Django 创建动态网页的基础知识。

2.1 第一个 Django 驱动的页面：Hello World

首先，我们来创建一个网页，输出著名的示例消息："Hello World"。如果不使用 Web 框架，你可以直接在一个文本文件中输入"Hello World"，把它命名为 `hello.html`，然后上传到 Web 服务器中的某个目录里。注意，在这个过程中要为那个网页指定两个重要信息：网页的内容（字符串 `"Hello world"`）和 URL（如 `http://www.example.com/hello.html`）。在 Django 中也要指定这两个信息，但是方式不同。页面的内容由视图函数（view function）生成，URL 在 URL 配置（URLconf）中指定。先来编写生成"Hello World"的视图函数。

2.1.1 第一个视图

在前一章创建的 `mysite` 目录中新建一个空文件，名为 `views.py`。这个模块用于保存本章编写的视图。生成"Hello World"的视图很简单。下面是完整的视图函数，以及导入语句。请把下述代码输入到 `views.py` 文件中。

```
from django.http import HttpResponse

def hello(request):
    return HttpResponse("Hello world")
```

我们来逐行分析一下这段代码：

- 首先，从 `django.http` 模块中导入 `HttpResponse` 类。导入这个类是因为后面的代码要使用。
- 然后，定义一个名为 `hello` 的函数，这是视图函数。视图函数至少有一个参数，按约定，名为 `request`。这是一个对象，包含触发这个视图的 Web 请求的信息，是 `django.http.HttpRequest` 类的实例。

这里，我们没有用到 `request`，但是必须作为第一个参数传给视图。注意，视图函数的名称没有关系，无需使用特定的方式命名，Django 能识别它。我们把这个视图命名为 `hello`，因为这个名称能明确表明视图的作用。如果愿意，也可以命名为 `hello_wonderful_beautiful_world`，或其他任何名称。2.1.2 节会说明 Django 查找这个函数的方式。

这个函数的定义体只有一行代码：返回使用文本 `"Hello world"` 实例化的 `HttpResponse` 对象。

这里的主要知识点是，视图就是普通的 Python 函数，它的第一个参数是 `HttpRequest` 对象，返回值是一个 `HttpResponse` 实例。Python 函数要想变成 Django 视图，必须做这两件事。（也有例外，后文说明。）

2.1.2 第一个 URL 配置

如果现在运行 `python manage.py runserver`，看到的仍然是"Welcome to Django"消息，"Hello World"视图完全

没有踪影。这是因为 mysite 项目还不知道有 hello 视图的存在，我们要明确告诉 Django 在某个 URL 上激活这个视图。还以发布静态 HTML 文件为例说明，现阶段相当于创建好了 HTML 页面，但是还没有把它上传到服务器中的某个目录里。

若想把视图函数与特定的 URL 对应起来，要使用 URL 配置（URLconf）。URL 配置相当于 Django 驱动的网站的目录。简单来说，URL 配置把 URL 映射到相应的视图函数上。我们以这种方式告诉 Django，“访问这个 URL 时调用这些代码，访问那个 URL 时调用那些代码”。

例如，有人访问 /foo/ 这个 URL 时，调用 views.py 模块中的 foo_view() 视图函数。前一章执行 djangoadmin startproject 时自动创建了一个 URL 配置：urls.py 文件。

这个文件的默认内容如下所示：

```
"""mysite URL Configuration

The urlpatterns list routes URLs to views. For more information please see:
    https://docs.djangoproject.com/en/1.8/topics/http/urls/
Examples:
Function views
    1. Add an import:  from my_app import views
    2. Add a URL to urlpatterns:  url(r'^$', views.home, name='home')
Class-based views
    1. Add an import:  from other_app.views import Home
    2. Add a URL to urlpatterns:  url(r'^$', Home.as_view(), name='home')
Including another URLconf
    1. Add an import:  from blog import urls as blog_urls
    2. Add a URL to urlpatterns:  url(r'^blog/', include(blog_urls))
"""
from django.conf.urls import include, url
from django.contrib import admin

urlpatterns = [
    url(r'^admin/', include(admin.site.urls)),
]
```

把文件顶部的文档注释去掉，剩下的是 URL 配置的精华：

```
from django.conf.urls import include, url
from django.contrib import admin

urlpatterns = [
    url(r'^admin/', include(admin.site.urls)),
]
```

我们来逐行分析这段代码：

- 第一行从 django.conf.urls 模块中导入两个函数：include，用于导入另一个 URL 配置模块；url，使用正则表达式模式匹配浏览器中的 URL，把它映射到 Django 项目中的某个模块上。
- 第二行从 django.contrib 模块中导入 admin 函数。这个函数传给 include 函数，加载 Django 管理后台的 URL。
- 第三行是 urlpatterns，即 url() 实例列表。

这里主要要注意的是 urlpatterns 变量，Django 期望 URL 配置模块中有这个变量。它负责定义 URL 与处理

URL 的代码之间的映射。在 URL 配置中添加 URL 和视图的方式是，把 URL 模式映射到视图函数上。添加 hello 视图的方式如下：

```
from django.conf.urls import include, url
from django.contrib import admin
from mysite.views import hello
urlpatterns = [
    url(r'^admin/', include(admin.site.urls)),
    url(r'^hello/$', hello),
]
```

我们做了两处修改：

- 首先，从 mysite/views.py（在 Python 导入句法中要使用 mysite.views）模块中导入 hello 视图。（假定 mysite/views.py 在 Python 路径中。）
- 然后，把 url(r'^hello/$', hello)，这行代码添加到 urlpatterns 中。我们添加的这行代码称为一个 URL 模式（URLpattern）。url() 函数告诉 Django 如何处理我们配置的 URL。第一个参数是模式匹配字符串（一个正则表达式，稍后说明），第二个参数是模式使用的视图函数。url() 还有一个可选参数，在第 7 章详述。

这里还有一个重要的细节：正则表达式字符串前面的 'r' 字符。它的目的是告诉 Python，那是“原始字符串”，不要解释里面的反斜线。

在常规的 Python 字符串中，反斜线用于转义特殊的字符，例如 "\n" 这个字符串是一个字符，表示换行。如果添加 r，标记为原始字符串，Python 不会转义；因此，"r'\n'" 是两个字符组成的字符串，一个是反斜线，一个是小写字母 n。

Python 中的反斜线与正则表达式中的反斜线有冲突，因此在 Django 中定义正则表达式时最好使用原始字符串。

综上，我们告诉 Django，对 /hello/ URL 的请求应该由 hello 视图函数处理。

URL 模式的句法需要说明一下，因为你或许不能立即理解。我们想匹配的是 /hello/ URL，但是用的模式与想象中有点区别。原因如下：

- Django 在检查 URL 模式时会把入站 URL 前面的斜线去掉。因此，URL 模式中没有前导斜线。一开始可能觉得这不符合直觉，但是这样做能简化一些事情，例如把 URL 配置引入其他 URL 配置（参阅第 7 章）。
- 模式中包含一个脱字符号（^）和一个美元符号（$）。它们在正则表达式中有特殊的意义：脱字符号表示“在字符串的开头匹配模式”，美元符号的意思是“在字符串的结尾匹配模式”。

  这个概念最好通过实例说明。如果使用 ^hello/ 模式（末尾没有美元符号），那么任何以 /hello/ 开头的 URL 字符串都能匹配，例如 /hello/foo 和 /hello/bar，而不只是 /hello/。

  类似地，如果开头没有脱字符号（即 hello/$），Django 会匹配任何以 hello/ 结尾的 URL，例如 /foo/bar/hello/。

  如果只使用 hello/，没有脱字符号或美元符号，那么任何包含 hello/ 的 URL 都匹配，例如 /foo/hello/bar。

  因此，我们使用脱字符号和美元符号确保只匹配 /hello/ 这个 URL——不多不少。大多数 URL 模式都以脱字符号开头、以美元符号结尾，不过可以灵活使用，匹配复杂的 URL。

你可能想知道，如果有人请求 /hello URL（末尾没有斜线）会发生什么。因为我们指定的 URL 模式要求有末尾的斜线，因此那个 URL 不匹配。然而，默认情况下，如果请求的 URL 不匹配任何 URL 模式，而且末尾

没有斜线，那么 Django 会把它重定向到末尾带斜线的 URL。（这个行为由 Django 的 `APPEND_SLASH` 设置管理，参见附录 D。）

关于这个 URL 配置，还有一件事要注意：我们以对象的形式传入 `hello` 视图函数，而没有调用函数。这是 Python（以及其他动态语言）的一个关键特性：函数是一等对象，可以像其他变量那样传递。很棒吧！

为了测试对 URL 配置的更改，像第 1 章那样启动 Django 开发服务器：运行 `python manage.py runserver` 命令。（如果开发服务器一直运行着也没关系，它能自动检测到 Python 代码的变化，按需重新加载，因此改动后无需重启服务器。）开发服务器运行在 `http://127.0.0.1:8000/` 地址上，因此要在 Web 浏览器中访问 `http://127.0.0.1:8000/hello/`。你应该能看到文本“Hello World”，即那个 Django 视图的输出（图 2-1）。

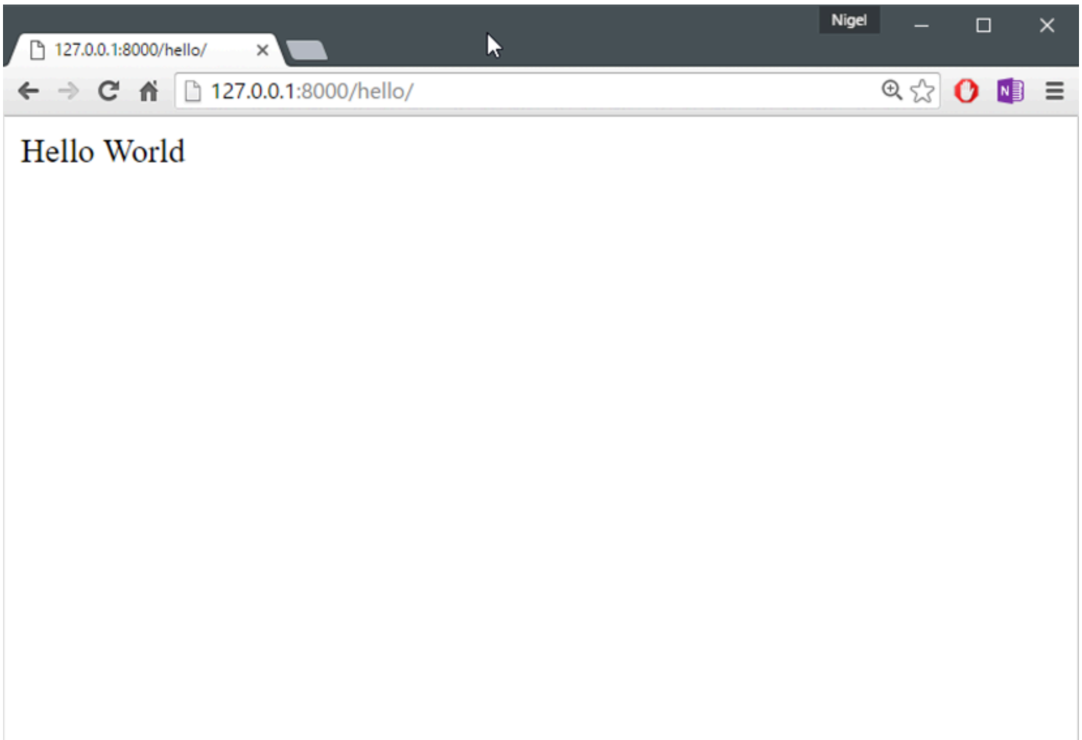


图 2-1：好耶！我们的第一个 Django 视图

2.1.3 正则表达式

正则表达式（regular expression，简称 regexes）是指定文本模式的简洁方式。Django 的 URL 配置允许使用任何正则表达式匹配复杂的 URL，但是实际上只会使用部分符号。表 2-1 列出了常用的符号。

表 2-1：常用的正则表达式符号

符号	匹配的内容
`.`（点号）	单个字符
`\d`	单个数字
`[A-Z]`	A-Z（大写）之间的单个字母
`[a-z]`	a-z（小写）之间的单个字母
`[A-Za-z]`	a-z（不区分大小写）之间的单个字母

（续）

符号	匹配的内容
+	一个或多个前述表达式（例如，\d+ 匹配一个或多个数字）
[^/]+	一个或多个字符，直到遇到斜线（不含）
?	零个或一个前述表达式（例如，\d? 匹配零个或一个数字）
*	零个或多个前述表达式（例如，\d* 匹配零个、一个或多个数字）
{1,3}	介于一个到三个之间（含）的前述表达式（例如，\d{1,3} 匹配一个、两个或三个数字）

正则表达式的详细说明参阅 Python 正则表达式文档。

2.1.4 关于 404 错误的简要说明

现在，URL 配置只定义了一个 URL 模式，即处理 `/hello/` URL 请求的那个。那么，如果请求其他 URL 会发生什么呢？为了查明，启动 Django 开发服务器，然后访问 `http://127.0.0.1:8000/goodbye/`。你应该会看到“Page not found”消息（图 2-2）。Django 之所以显示这个消息，是因为 URL 配置中没有定义你请求的 URL。

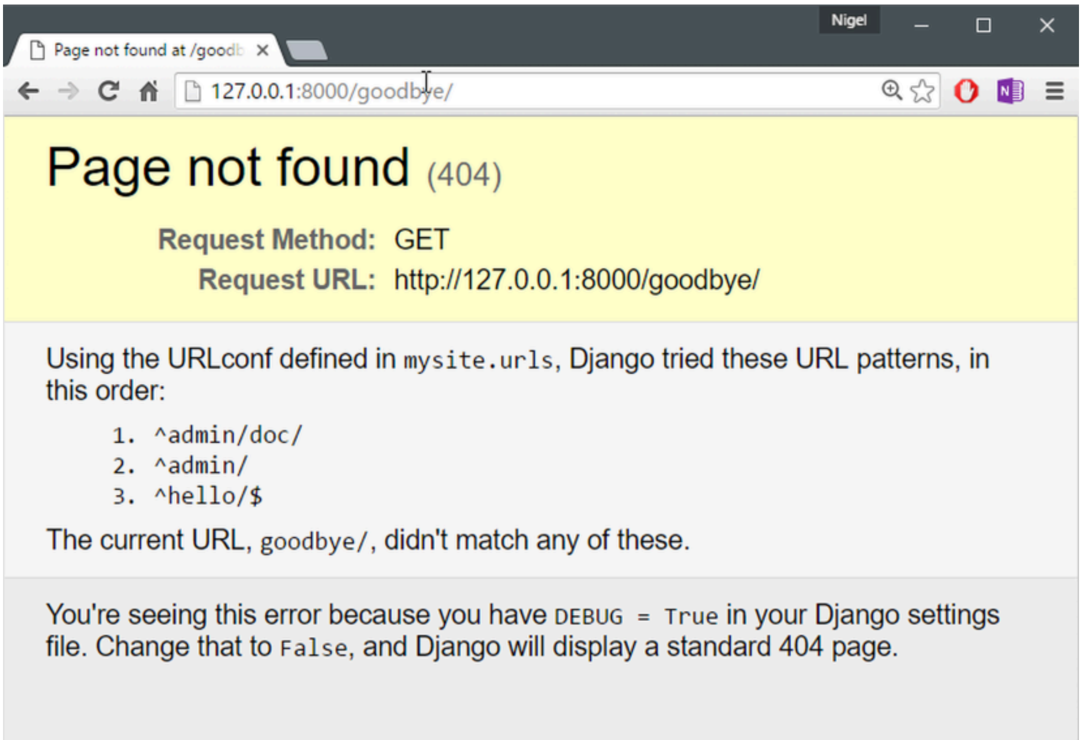


图 2-2：Django 的 404 页面

这个页面除了显示 404 错误消息之外，还给出了其他信息。它会告诉你 Django 使用的是哪个 URL 配置，以及那个配置里的各个模式。根据这些信息你应该能判断为什么所请求的 URL 会返回 404 错误。

显然，这些是敏感信息，只供 Web 开发者查看。线上网站不应该公开显示这些信息。鉴于此，“Page not found”页面仅当 Django 项目处于调试模式时才会显示。

后文会说明如何解除调试模式。现在，你只需知道，创建 Django 项目后，它就处于调试模式。如果不在调试模式中，Django 会输出其他的 404 响应。

2.1.5 关于网站根地址的简要说明

据前一节所述，如果访问网站根地址（`http://127.0.0.1:8000/`），你会看到一个 404 错误消息。Django 不会自作主张在网站根地址上添加页面——根地址没什么特别的。

你要像 URL 配置中的其他条目一样，为根地址指定一个 URL 模式。匹配网站根地址的 URL 模式不太直观，需要说明一下。

准备为网站根地址实现视图时，使用的 URL 模式是 `^$`，即匹配空字符串。例如：

```
from mysite.views import hello, my_homepage_view

urlpatterns = [
    url(r'^$', my_homepage_view),
    # ...
```

2.1.6 Django 处理请求的过程

继续编写第二个视图函数之前，我们停一下，稍微了解 Django 的运作机制。你在 Web 浏览器中访问 `http://127.0.0.1:8000/hello/`，看到“Hello world”消息，在这个过程中 Django 在背后做了什么呢？一切都从设置文件开始。

运行 `python manage.py runserver` 命令时，`manage.py` 脚本在内层 `mysite` 目录中寻找名为 `settings.py` 的文件。这个文件中保存着当前 Django 项目的全部配置，各个配置的名称都是大写的，例如 `TEMPLATE_DIRS`、`DATABASES`，等等。其中最重要的设置是 `ROOT_URLCONF`。它告诉 Django，网站的 URL 配置在哪个 Python 模块中。

记得吗？运行 `django-admin startproject` 命令时创建了 `settings.py` 和 `urls.py` 文件。自动生成的 `settings.py` 文件中包含 `ROOT_URLCONF` 设置，指向自动生成的 `urls.py` 文件。打开 `settings.py` 文件，你自己看一下。应该会看到下述设置：

```
ROOT_URLCONF = 'mysite.urls'
```

这个模块对应的文件是 `mysite/urls.py`。收到针对某个 URL（假如是 `/hello/`）的请求时，Django 加载 `ROOT_URLCONF` 设置指定的 URL 配置；然后按顺序检查 URL 配置中的各个 URL 模式，依次与请求的 URL 比较，直到找到匹配的模式为止。

找到匹配的模式之后，调用对应的视图函数，并把一个 `HttpRequest` 对象作为第一个参数传给视图。（后文会详述 `HttpRequest`。）如我们编写的第一个视图所示，视图函数必须返回一个 `HttpResponse` 对象。

随后，余下的工作交给 Django 处理：把那个 Python 对象转换成正确的 Web 响应，并且提供合适的 HTTP 首部和主体（即网页的内容）。综上：

1. 请求 `/hello/`。
2. Django 查看 `ROOT_URLCONF` 设置，找到根 URL 配置。
3. Django 比较 URL 配置中的各个 URL 模式，找到与 `/hello/` 匹配的那个。
4. 如果找到匹配的模式，调用对应的视图函数。
5. 视图函数返回一个 `HttpResponse` 对象。

6. Django 把 HttpResponse 对象转换成正确的 HTTP 响应，得到网页。

现在，我们知道如何编写 Django 驱动的页面了。整个过程十分简单，只需编写视图函数，然后在 URL 配置中与 URL 配对。

2.2 第二个视图：动态内容

“Hello World”视图是为了说明 Django 的基本运作方式，但那不是动态网页，因为页面的内容始终不变。每次访问 /hello/，看到的都是相同的内容，好似一个静态 HTML 文件。

这一次，我们要创建一些动态内容，在网页中显示当前日期和时间。这是不错的第二步，因为不涉及数据库或用户输入，只是输出服务器的内部时钟。这个页面不比“Hello World”强大多少，但是通过它能说明一些新概念。这个视图要做两件事：计算当前日期和时间，然后返回包含那个值的 HttpResponse 对象。如果用过 Python，你应该知道 Python 有个 datetime 模块，用于计算日期。用法如下：

```
>>> import datetime
>>> now = datetime.datetime.now()
>>> now
datetime.datetime(2015, 7, 15, 18, 12, 39, 2731)
>>> print (now)
2015-07-15 18:12:39.002731
```

很简单，而且不用麻烦 Django。这就是常规的 Python 代码。（我会强调“常规的 Python”代码和 Django 专属的代码。在学习 Django 的过程中，希望你能学会把知识运用到 Django 之外的 Python 项目中。）若想在 Django 视图中显示当前日期和时间，我们只需把 datetime.datetime.now() 语句放到一个视图中，然后返回一个 HttpResponse 对象。参照下述代码更新 views.py：

```
from django.http import HttpResponse
import datetime

def hello(request):
    return HttpResponse("Hello world")

def current_datetime(request):
    now = datetime.datetime.now()
    html = "It is now %s." % now
    return HttpResponse(html)
```

为了理解 current_datetime 视图，下面逐行分析所做的改动：

- 我们在模块顶部添加了 import datetime，这样才能计算日期。
- 新添加的 current_datetime 函数计算当前日期和时间，得到的结果是一个 datetime.datetime 对象，存储在局部变量 now 中。
- 视图函数中的第二行代码使用 Python 的格式字符串功能构建一个 HTML 响应。字符串中的 %s 是占位符，字符串后面的百分号指明，“把前述字符串中的 %s 替换成 now 变量的值”。严格来说，now 变量的值是一个 datetime.datetime 对象，而非字符串，但是 %s 格式字符会把它转换成字符串表示形式，得到的结果类似于 "2015-07-15 18:12:39.002731"。最终得到的 HTML 字符串是这样的："It is now 2015-07-15 18:12:39.002731."。
- 最后，返回包含那个 HTML 字符串的 HttpResponse 对象——这与 hello 视图所做的一样。

把相关代码添加到 views.py 中之后，要在 urls.py 中添加 URL 模式，告诉 Django，哪个 URL 使用这个视图

处理。这里可以使用 /time/。

```
from django.conf.urls import include, url
from django.contrib import admin
from mysite.views import hello, current_datetime

urlpatterns = [
    url(r'^admin/', include(admin.site.urls)),
    url(r'^hello/$', hello),
    url(r'^time/$', current_datetime),
]
```

我们对 URL 配置做了两处改动。其一，在顶部导入了 current_datetime 函数；其二，这是较为重要的，添加一个 URL 模式，把 /time/ URL 映射到那个新视图上。能理解吧？编写视图并且更新 URL 配置之后，运行 runserver 命令，然后在浏览器中访问 http://127.0.0.1:8000/time/。你应该看到当前日期和时间。如果看到的不是本地时间，有可能是因为在 settings.py 中把默认时区设为 'UTC' 了。

2.3 URL 配置和松耦合

现在是个好时机，我们要强调 URL 配置以及 Django 背后的一个重要哲学：松耦合。简单来说，松耦合是一种软件开发方式，其价值在于让组件可以互换。如果两部分代码之间是松耦合的，那么改动其中一部分对另一部分的影响很小，甚至没有影响。

Django 的 URL 配置就很好地运用了这个原则。在 Django Web 应用中，URL 定义与所调用的视图函数之间是松耦合的，即某个功能使用哪个 URL 与视图函数的实现本身放在两个地方。

以 current_datetime 视图为例。如果想把这个功能对应的 URL 从 /time/ 移到 /current-time/，只需修改 URL 配置，视图则不用动。同样，对视图函数的逻辑所做的修改对开放这一功能的 URL 没有影响。此外，如果想在多个 URL 上开放当前日期功能，可以编辑 URL 配置，而无需改动视图代码。

在下述示例中，current_datetime 可以通过两个 URL 访问。我们是故意这么做的，但是其中涉及的知识能用得到。

```
urlpatterns = [
    url(r'^admin/', include(admin.site.urls)),
    url(r'^hello/$', hello),
    url(r'^time/$', current_datetime),
    url(r'^another-time-page/$', current_datetime),
]
```

URL 配置和视图之间就是松耦合的。在本书后续内容中我会不断指出运用这个重要原则的地方。

2.4 第三个视图：动态 URL

在 current_datetime 视图中，页面的内容（当前日期和时间）是动态的，但 URL 是静态的（/time/）。可是，在多数动态 Web 应用中，URL 中会包含参数，影响页面的输出。比如说，在线书店会为每本书分配一个 URL，例如 /books/243/ 和 /books/81196/。下面创建第三个视图，以一定的偏移量显示当前日期和时间。我们的目标是让 /time/plus/1/ 页面显示一小时之后的日期和时间，让 /time/plus/2/ 页面显示两小时之后的日期和时间，让 /time/plus/3/ 页面显示三小时之后的日期和时间，以此类推。新手可能想分别为各个偏移量编写一个视图函数，然后在 URL 配置中这样定义模式：

```
urlpatterns = [
```

```
        url(r'^time/$', current_datetime),
        url(r'^time/plus/1/$', one_hour_ahead),
        url(r'^time/plus/2/$', two_hours_ahead),
        url(r'^time/plus/3/$', three_hours_ahead),
    ]
```

显然，这样想是错的。

这样做不仅要重复编写视图函数，而且应用只能支持预先定义的偏移范围（一小时、两小时或三小时）。

如果想创建显示四小时之后的时间的页面，必须再编写一个视图，并且添加一行 URL 配置——重复更多了！

那么，为了处理任意的偏移量，应该怎么设计应用呢？问题的关键是，使用通配 URL 模式。前面说过，URL 模式是正则表达式，因此我们可以使用 `\d+` 模式匹配一个或多个数字：

```
    urlpatterns = [
        # ...
        url(r'^time/plus/\d+/$', hours_ahead),
        # ...
    ]
```

（`# …` 的意思是其他 URL 模式被省略了。）这个 URL 模式能匹配 `/time/plus/2/`、`/time/plus/25/`，甚至是 `/time/plus/100000000000/`。但是，这么大的偏移量不太切合实际，我们将为偏移量设个合理的最大值。

这里，我们想把最大偏移量设为 99 小时，因此要限制只能使用一个或两个数字。使用正则表达式句法来表示，就是 `\d{1,2}`：

```
    url(r'^time/plus/\d{1,2}/$', hours_ahead),
```

现在，我们配置好了通配 URL，下面要把通配数据传给视图函数，以便在一个视图函数中处理任意的偏移量。为此，我们在 URL 模式中放一对圆括号，把想保存的数据括起来。在这个示例中，我们要保存 URL 中的数字，因此要在 `\d{1,2}` 两侧放置圆括号，如下所示：

```
    url(r'^time/plus/(\d{1,2})/$', hours_ahead),
```

如果你熟悉正则表达式，应该能理解这么做的作用；我们使用圆括号从匹配的文本中捕获数据。最终的 URL 配置，以及前面两个视图的配置，如下所示：

```
    from django.conf.urls import include, url
    from django.contrib import admin
    from mysite.views import hello, current_datetime, hours_ahead
    urlpatterns = [
        url(r'^admin/', include(admin.site.urls)),
        url(r'^hello/$', hello),
        url(r'^time/$', current_datetime),
        url(r'^time/plus/(\d{1,2})/$', hours_ahead),
    ]
```

提示

如果你用过其他 Web 开发平台，可能会想，“使用查询字符串参数吧！”，比如 /time/plus?hours=3；此时，偏移量通过 URL 的查询字符串获取（? 之后的部分）。

在 Django 中也能这么做（详情参见第 7 章），但是 Django 的核心哲学之一是，URL 应该美观。/time/plus/3/ 这样的 URL 更加简洁明了，更加易于阅读，更加易于大声读出来……而且比使用查询字符串的版本美观得多。美观的 URL 是高质量 Web 应用的一个特色。Django 的 URL 配置系统鼓励使用美观的 URL，而且为此提供了简易的方式。

设好 URL 之后，下面编写 hours_ahead 视图。这个视图与前面编写的 current_datetime 视图很像，唯有一处不同：要接收额外的参数，即偏移的小时数。hours_ahead 视图的代码如下：

```
from django.http import Http404, HttpResponse
import datetime

def hours_ahead(request, offset):
    try:
        offset = int(offset)
    except ValueError:
        raise Http404()
    dt = datetime.datetime.now() + datetime.timedelta(hours=offset)
    html = "In %s hour(s), it will be  %s.
            " % (offset, dt)
    return HttpResponse(html)
```

下面来分析这段代码。

hours_ahead 视图函数有两个参数：request 和 offset。

- request 是一个 HttpRequest 对象，这与 hello 和 current_datetime 两个视图一样。我要再次强调，每个视图的第一个参数都是 HttpRequest 对象。
- offset 是 URL 模式中那对圆括号捕获的字符串。如果请求的 URL 是 /time/plus/3/，那么偏移量是字符串 '3'；如果请求的 URL 是 /time/plus/21/，那么偏移量是字符串 '21'。注意，捕获的值始终是 Unicode 对象，而不是整数，即便捕获的字符串中只有数字（如 '21'）也不是。

我决定把那个变量命名为 offset，不过你可以根据自己的喜好命名，只要是有效的 Python 标识符就行。变量使用什么名称没关系，重要的是要作为第二个参数（放在 request 之后）传给视图函数。（除了位置参数之外，URL 配置还可以使用关键字参数。详情参见第 7 章。）

在视图函数中，我们首先调用 int() 处理 offset。这么做的目的是把 Unicode 字符串转换成整数。

注意，如果传给 int() 函数的值不能转换成整数（如字符串 "foo"），Python 会抛出 ValueError 异常。在这个示例中，如果遇到 ValueError 异常，我们抛出 django.http.Http404 异常，显示“Page not found”404 错误（你可能猜到了）。

聪明的读者可能会想，怎么可能出现 ValueError 呢？毕竟 URL 模式中的正则表达式（(\d{1,2})）只能捕获数字，因此 offset 变量的值只可能是由数字组成的字符串。答案是，无法保证。URL 模式提供的输入验证虽然有用，但是并不严谨，因此我们要在视图函数中检查会不会抛出 ValueError 异常，以防意外。实现视图函数时最好别对参数做任何假设。松耦合，还记得吗？

视图函数的下一行计算当前日期和时间，并且加上相应的偏移量。current_datetime 视图用过 date-

time.datetime.now()；这里的新知识是，表示日期和时间的两个 datetime.datetime 对象可以做算术运算。我们把计算结果存储在 dt 变量中。

这一行还表明了调用 int() 处理 offset 的原因：datetime.timedelta 的 hour 参数必须是整数。

接下来，与 current_datetime 视图一样，构建这个视图函数的 HTML 输出。与之前不同的是，这里使用的 Python 格式字符串能处理两个值，而不是一个。因此，格式字符串中有两个 %s 符号，而且要插入的值是一个元组：(offset, dt)。

最后，返回包含那个 HTML 的 HttpResponse 对象。

视图函数写好了，URL 也配置好了，现在启动 Django 开发服务器（如果没在运行中），然后访问 http://127.0.0.1:8000/time/plus/3/，确认是否正常。

然后访问 http://127.0.0.1:8000/time/plus/5/。

再访问 http://127.0.0.1:8000/time/plus/24/。

最后，访问 http://127.0.0.1:8000/time/plus/100/，确保 URL 配置中的模式只能匹配一个或两个数字。此时，Django 应该显示“Page not found”页面，与 2.1.4 节看到的一样。

http://127.0.0.1:8000/time/plus/（没有偏移量）也应该返回 404 错误。

2.5 Django 精美的错误页面

花点时间欣赏一下我们构建的这个 Web 应用……然后，搞点破坏。我们要把 hours_ahead 视图中 offset = int(offset) 那几行注释掉，故意在 views.py 文件中引入一个 Python 错误：

```
def hours_ahead(request, offset):
    # try:
    #     offset = int(offset)
    # except ValueError:
    #     raise Http404()
    dt = datetime.datetime.now() + datetime.timedelta(hours=offset)
    html = "In %s hour(s), it will be %s.
            " % (offset, dt)
    return HttpResponse(html)
```

启动开发服务器，然后访问 /time/plus/3/。你会看到一个错误页面，里面有相当多的信息，包括最顶部显示的 TypeError 消息："unsupported type for timedelta hours component: str"（图 2-3）。

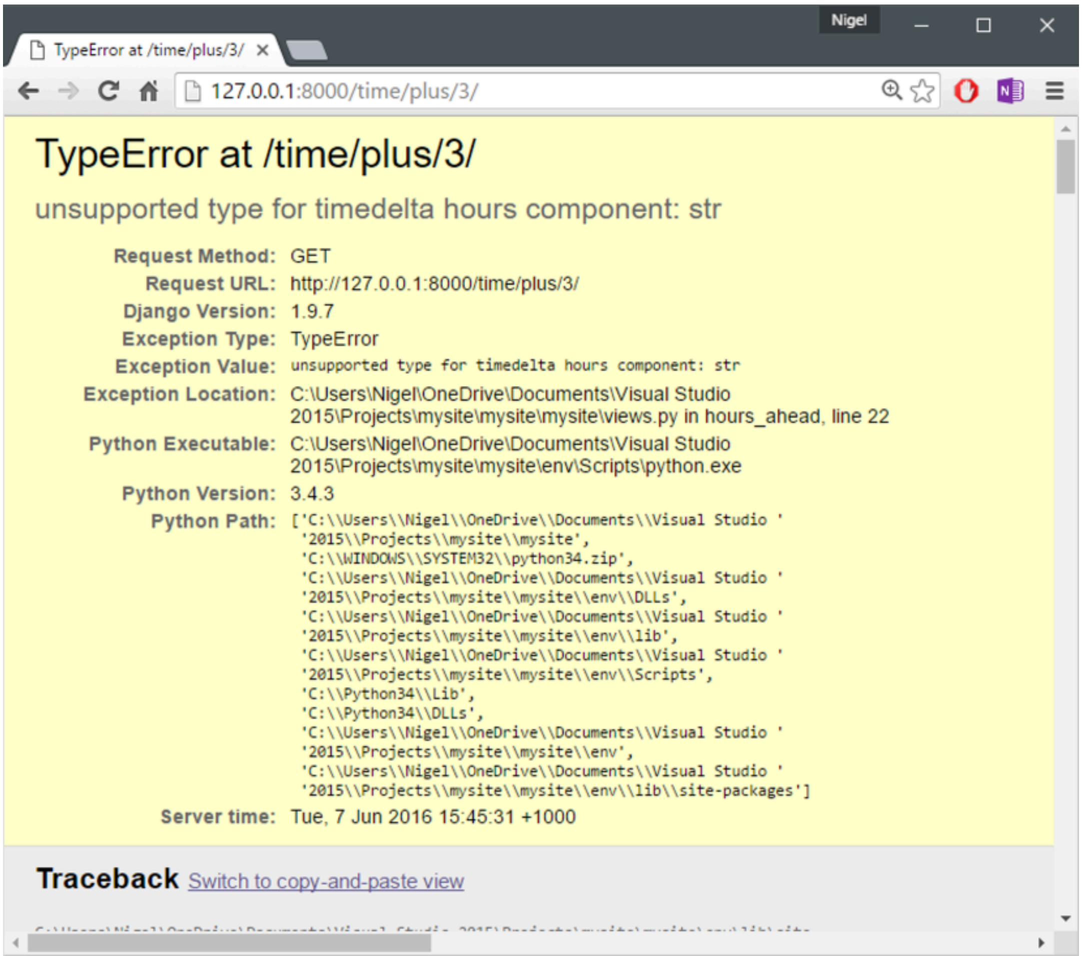


图 2-3：Django 的错误页面

怎么回事？`datetime.timedelta` 函数期望 `hours` 参数是整数，而把 `offset` 转换成整数的代码被我们注释掉了。因此，`datetime.timedelta` 抛出 `TypeError` 异常。这种小问题每个程序员都会遇到。

这个示例的目的是说明 Django 的错误页面。花点时间浏览错误页面，熟悉它给出的各部分信息。值得关注的信息有：

- 页面顶部是异常的关键信息：异常的类型和消息（这里是 `"unsupported type"`），抛出异常的文件，以及所在的行号。
- 关键信息下面是异常的完整 Python 调用跟踪。这与 Python 命令解释器给出的标准调用跟踪类似，不过能与之交互。对栈里的每一层（“帧”），Django 都会显示文件名、函数/方法名、行号和那一行的源码。
- 点击源码所在的行（深灰色），会显示前后数行，为你提供上下文。点击每一帧下面的“Local vars”可以查看那一帧抛出异常时所有局部变量及其值。这些信息能给调试提供极大的帮助。
- 注意“Traceback”标题旁边的“Switch to copy-and-paste view”文本。点击那些文本之后，调用跟踪会切换到另一个版本，以便复制粘贴。如果想把异常的调用跟踪分享给他人（例如 Django IRC 或 Django 用户邮件列表中友善的人），寻求技术支持，可以使用这个版本。
- 切换之后，底部会显示一个“Share this traceback on a public Web site”按钮，只需点击一下就能把调用跟踪发布到网上。点击那个按钮之后，调用跟踪会发布到 dpaste 网站中，而且会得到一个 URL，便于分享给其他人。

- 接下来是“Request information”（请求信息）部分，里面包含大量与导致错误的入站 Web 请求有关的信息：GET 和 POST 信息、cookie 值，以及元信息，如 CGI 首部。附录 F 有一份关于请求对象中所含信息的完整参考。
- “Request information”部分下面是“Settings”（设置）部分，列出 Django 当前的全部设置。各个设置在附录 D 中详述。

在某些情况下（如模板句法错误），Django 的错误页面还会显示更多信息。后文讨论 Django 的模板系统时会说明这种情况。现在，把 offset = int(offset) 相关的那几行代码的注释去掉，让视图函数能正确运行。

如果你喜欢使用 print 语句调试，会发现 Django 的错误页面特别有用。

在视图的任意位置临时插入 assert False 语句，触发错误。然后，查看局部变量和程序的状态。下面以 hours_ahead 视图为例：

```
def hours_ahead(request, offset):
    try:
        offset = int(offset)
    except ValueError:
        raise Http404()
    dt = datetime.datetime.now() + datetime.timedelta(hours=offset)
    assert False
    html = "In %s hour(s), it will be %s.
            " % (offset, dt)
    return HttpResponse(html)
```

最后要说一点。显然，这些信息大都是敏感的，暴漏了内部的 Python 代码和 Django 配置，因此别天真地以为可以在网上公开显示。不怀好意的人可以利用这些信息做逆向工程，然后在你的 Web 应用中做些坏事。鉴于此，仅当 Django 项目处于调试模式时才会显示错误页面。第 13 章会说明如何解除调试模式。现在，你只需知道，Django 项目一开始都自动处于调试模式中。（听着耳熟？本章前面所述的“Page not found”错误页面也是如此。）

2.6 接下来

目前，我们编写的视图函数都直接在 Python 代码中硬编码 HTML。我这么做是为了简化，以便讨论核心概念。但在实际开发中，这是十分糟糕的做法。Django 自带了一套简单却强大的模板引擎，能把页面的设计和底层代码隔开。下一章将讨论 Django 的模板引擎。

第 3 章 Django 模板

你可能注意到了，前一章在示例视图中返回文本的方式有些特别，即在 Python 代码中硬编码 HTML，如下所示：

```
def current_datetime(request):
    now = datetime.datetime.now()
    html = "It is now %s." % now
    return HttpResponse(html)
```

这样做虽然便于说明视图的工作方式，但是直接在视图中硬编码 HTML 不是个好主意。原因如下：

- 只要想修改页面的设计就要修改 Python 代码。网站的设计肯定比底层的 Python 代码变化频繁，因此如果无需修改 Python 代码就能改变设计，那多方便。
- 这只是十分简单的示例。网页模板往往包含几百行 HTML 和脚本。在这样的混乱中排错和诊断程序代码简直是噩梦。（我说的是 PHP 吗？）
- 编写 Python 代码和设计 HTML 是两件不同的事，多数专业的 Web 设计团队会把这两件事交给不同的人做（甚至不同的部门）。设计师和 HTML/CSS 程序员完成工作不应该需要编辑 Python 代码。
- 如果编写 Python 代码的程序员和设计模板的设计师能同时工作，而不用等到一个人编辑好包含 Python 和 HTML 的文件之后再交给下一个人，工作效率能得到提升。

鉴于此，把页面的设计与 Python 代码分开，结果更简洁，也更易于维护。为此，我们可以使用 Django 的模板系统。本章就讨论这个话题。

3.1 模板系统基础

Django 模板是一些文本字符串，作用是把文档的表现与数据区分开。模板定义一些占位符和基本的逻辑（模板标签），规定如何显示文档。通常，模板用于生成 HTML，不过 Django 模板可以生成任何基于文本的格式。

> **Django 模板背后的哲学**
>
> 如果你有编程背景，或者用过把程序代码直接混入 HTML 的语言，要记住一件事：Django 的模板可不是把 Python 代码嵌入 HTML 这么简单。模板系统的设计目的是呈现表现，而不是程序逻辑。

下面看个简单的模板示例。这个 Django 模板描述一个 HTML 页面，感谢用户在公司的网站中购物。你可以把它理解为一个通函。

```
<html>
<head>
    <title>Ordering notice</title>
</head>
<body>
```

```
    <h1>Ordering notice</h1>

    <p>Dear {{ person_name }},</p>

    <p>Thanks for placing an order from {{ company }}. It's scheduled to ship on {{
ship_date|date:"F j, Y" }}.</p>
    <p>Here are the items you've ordered:</p>
    <ul>
        {% for item in item_list %}
        <li>{{ item }}</li>{% endfor %}
    </ul>

    {% if ordered_warranty %}
    <p>Your warranty information will be included in the packaging.</p>
    {% else %}
    <p>
        You didn't order a warranty, so you're on your own when
        the products inevitably stop working.
    </p>
    {% endif %}

    <p>Sincerely,<br />{{ company }}</p>

</body>
</html>
```

这个模板的大部分内容是 HTML，内含一些变量和模板标签。下面详细说明。

- 两对花括号包围的文本（如 `{{ person_name }}`）是变量，意思是“把指定变量的值插入这里”。如何指定变量的值呢？稍后说明。
- 一对花括号和百分号包围的文本（如 `{% if ordered_warranty %}`）是模板标签。

  标签的定义相当宽泛：只要能让模板系统“做些事”的就是标签。
- 这个示例模板中有一个 `for` 标签（`{% for item in item_list %}`）和一个 `if` 标签（`{% if ordered_warranty %}`）。`for` 标签的作用与 Python 中的 `for` 语句很像，用于迭代序列中的各个元素。与你预期的一样，`if` 标签的作用是编写逻辑判断语句。

  这里，`if` 标签检查 `ordered_warranty` 变量的求值结果是不是 `True`。如果是，模板系统将显示 `{% if ordered_warranty %}` 和 `{% else %}` 之间的内容；如果不是，模板系统将显示 `{% else %}` 和 `{% endif %}` 之间的内容。注意，`{% else %}` 是可选的。
- 最后，这个模板的第二段包含一个过滤器，这是调整变量格式最为便利的方式。对这个示例中的 `{{ ship_date|date:"F j, Y" }}` 来说，我们把 `ship_date` 变量传给 `date` 过滤器，并且为 `date` 过滤器指定 `"F j, Y"` 参数。`date` 过滤器使用参数指定的格式格式化日期。过滤器使用管道符号（`|`）依附，类似于 Unix 管道。

Django 模板能访问多个内置的标签和过滤器，接下来的几节会讨论其中几个。附录 E 列出了全部标签和过滤器，你最好熟悉一下，知道有哪些可用。我们自己也可以创建过滤器和标签，参阅第 8 章。

3.2 使用模板系统

Django 系统经过配置后可以使用一个或多个模板引擎（如果不用模板，那就不用配置）。Django 自带了一个

内置的后端，用于支持自身的模板引擎，即 Django Template Language（DTL）。Django 1.8 还支持另一个流行的模板引擎，Jinja2。如果没有特别的理由更换后端，应该使用 DTL——如果编写的是可插入式应用，而且带有模板，更应该如此。Django 包含模板的 contrib 应用，如 django.contrib.admin，使用的就是 DTL。本章的所有示例都将使用 DTL。高级的模板话题，如配置第三方模板引擎，参阅第 8 章。在视图中使用 Django 模板之前，先稍微说明一下 DTL，了解它的工作方式。若想在 Python 代码中使用 Django 的模板系统，基本方式如下：

1. 以字符串形式提供原始的模板代码，创建 Template 对象。
2. 在 Template 对象上调用 render() 方法，传入一系列变量（上下文）。返回的是完全渲染模板后得到的字符串，模板中的变量和模板标签已经根据上下文求出值了。

用代码来说，上述过程如下：

```
>>> from django import template
>>> t = template.Template('My name is {{ name }}.')
>>> c = template.Context({'name': 'Nige'})
>>> print (t.render(c))
My name is Nige.
>>> c = template.Context({'name': 'Barry'})
>>> print (t.render(c))
My name is Barry.
```

接下来的几节详述各步。

3.2.1 创建 Template 对象

创建 Template 对象最简单的方式是直接实例化。Template 类在 django.template 模块中，构造方法接受一个参数，即原始的模板代码。下面在 Python 交互式解释器中通过代码说明。在第 1 章创建的 mysite 项目目录中输入 python manage.py shell，启动交互式解释器。

下面说明模板系统的一些基本知识：

```
>>> from django.template import Template
>>> t = Template('My name is {{ name }}.')
>>> print (t)
```

在交互式解释器中输入上述代码后会得到类似下面的输出：

```
<django.template.base.Template object at 0x030396B0>
```

在每次得到的输出中，0x030396B0 那部分是不同的，而且也无关紧要。那是 Python 的细节（如果一定想知道的话，我告诉你，那是 Template 对象的“标识”）。

创建 Template 对象时，模板系统会把原始的模板代码编译成内部使用的优化形式，为渲染做好准备。但是，如果模板代码中有句法错误，调用 Template() 会导致 TemplateSyntaxError 异常抛出：

```
>>> from django.template import Template
>>> t = Template('{% notatag %}')
Traceback (most recent call last):
File "", line 1, in ?
...
django.template.base.TemplateSyntaxError: Invalid block tag: 'notatag'
```

这里的“block tag”（块级标签）指的是 {% notatag %}。“块级标签”和“模板标签”是同一个事物。遇到下述各种情况时，模板系统都会抛出 TemplateSyntaxError：

- 无效标签
- 有效标签的无效参数
- 无效过滤器
- 有效过滤器的无效参数
- 无效模板句法
- 未关闭的标签（对需要结束标签的模板标签而言）

3.2.2 渲染模板

有了 Template 对象之后，可以为其提供上下文，把数据传给它。上下文就是一系列模板变量和相应的值。模板使用上下文填充变量，求值标签。在 Django 中，上下文使用 django.template 模块中的 Context 类表示。它的构造方法接受一个可选参数：一个字典，把变量名映射到值上。“填充”模板的方式是，在 Template 对象上调用 render() 方法，并传入上下文：

```
>>> from django.template import Context, Template
>>> t = Template('My name is {{ name }}.')
>>> c = Context({'name': 'Stephane'})
>>> t.render(c)
'My name is Stephane.'
```

旁注 3-1：一个特殊的 Python 提示符

如果你用过 Python，可能会想，为什么运行 python manage.py shell，而不是 python（或 python3）。这两个命令都能启动交互式解释器，但是 manage.py shell 命令有个关键区别：在启动解释器之前，告诉 Django 使用哪个设置文件。Django 的很多部分，包括模板系统，依赖于设置，如果不告诉 Django 使用哪个设置，你就无法使用它们。如果你好奇，我来说说这背后的原理。Django 查找一个名为 DJANGO_SETTINGS_MODULE 的环境变量，其值是导入 settings.py 的路径。例如，DJANGO_SETTINGS_MODULE 的值可能是 'mysite.settings'（假设 mysite 在 Python 路径中）。执行 python manage.py shell 命令时，它会为你设定 DJANGO_SETTINGS_MODULE。在这些示例中必须使用 python manage.py shell，否则 Django 会抛出异常。

3.3 字典和上下文

Python 字典是键和值的映射。Context 对象类似于字典，但是它有额外的功能，参阅第 8 章。

变量名必须以字母开头（A-Z 或 a-z），随后可以跟着任意多个字母、数字、下划线和点号。（点号是特殊的，稍后说明。）变量名区分大小写。下面是编译和渲染模板的示例，使用的模板类似于本章开头那个：

```
>>> from django.template import Template, Context
>>> raw_template = """<p>Dear {{ person_name }},</p>
...
... <p>
    Thanks for placing an order from {{ company }}. It's scheduled to
    ... ship on {{
```

```
ship_date|date:"F j, Y"
    }}.
</p>
...
... {% if ordered_warranty %}
... <p>Your warranty information will be included in the packaging.</p>
... {% else %}
... <p>
    You didn't order a warranty, so you're on your own when
    ... the products inevitably stop working.
</p>
... {% endif %}
...
... <p>Sincerely,<br />{{ company }}</p>"""
>>> t = Template(raw_template)
>>> import datetime
>>> c = Context({'person_name': 'John Smith',
...     'company': 'Outdoor Equipment',
...     'ship_date': datetime.date(2015, 7, 2),
...     'ordered_warranty': False})
>>> t.render(c)
"<p>Dear John Smith,</p>\n\n<p>Thanks for placing an order from Outdoor Equipment. It\
's scheduled to\nship on July 2,2015.</p>\n\n\n<p>You didn't order a warranty, so you\
're on your own when\nthe products inevitably stop working.</p>\n\n\n<p>Sincerely,<br\
 />Outdoor Equipment</p>"
```

- 首先，导入 `Template` 和 `Context` 类，二者都在 `django.template` 模块中。
- 把模板的原始文本保存在 `raw_template` 变量中。注意，我们使用三个引号标明那个字符串，这是因为它分为多行。使用单引号标明的字符串不能分成多行。
- 接下来，把 `raw_template` 传给 `Template` 类的构造方法，创建一个模板对象，名为 `t`。
- 从 Python 的标准库中导入 `datetime` 模块，因为下一个语句要使用。
- 然后，创建一个 `Context` 对象，名为 `c`。`Context` 构造方法接受一个 Python 字典，把变量名映射到值上。这里，我们把 `person_name` 设为 `'John Smith'`，把 `company` 设为 `'Outdoor Equipment'`，等等。
- 最后，在模板对象上调用 `render()` 方法，并且传入上下文。返回的是渲染后的模板，即把模板变量替换成变量的值，并且执行模板标签。注意，显示的是“You didn’t order a warranty”，因为 `ordered_warranty` 变量的求值结果是 `False`。还要注意日期，根据格式字符串 `F j, Y`，显示为 `July 2, 2015`。（稍后说明 `date` 过滤器的格式字符串。）

如果你刚接触 Python，可能会想，输出中为什么包含换行符（`\n`），而不直接换行。之所以这样，是因为 Python 交互式解释器的一个微妙之处：`t.render(c)` 返回一个字符串，而交互式解释器默认显示字符串的表示形式，而不是打印字符串的值。如果想让换行显示为真正的换行，而不是 `\n` 字符，使用 `print` 函数：`print (t.render(c))`。

以上就是 Django 模板系统的基本用法：编写模板字符串，创建 `Template` 对象，创建 `Context` 对象，然后调用 `render()` 方法。

3.3.1 多个上下文，同一个模板

得到 `Template` 对象之后，可以用其渲染多个上下文。例如：

```
>>> from django.template import Template, Context
>>> t = Template('Hello, {{ name }}')
>>> print (t.render(Context({'name': 'John'})))
Hello, John
>>> print (t.render(Context({'name': 'Julie'})))
Hello, Julie
>>> print (t.render(Context({'name': 'Pat'})))
Hello, Pat
```

像这样使用同一个模板渲染多个上下文，比分成多次创建 Template 对象再调用 render() 效率高：

```
# 不好
for name in ('John', 'Julie', 'Pat'):
t = Template('Hello, {{ name }}')
print (t.render(Context({'name': name})))

# 好
t = Template('Hello, {{ name }}')
for name in ('John', 'Julie', 'Pat'):
print (t.render(Context({'name': name})))
```

Django 解析模板的速度相当快。在背后，解析的大部分工作通过调用一个正则表达式完成。这与基于 XML 的模板系统有显著区别，XML 解析器会带来额外的消耗，从而导致渲染速度比 Django 的模板渲染引擎慢几个数量级。

3.3.2 上下文变量查找

目前所举的示例，传入上下文的都是简单的值，大多数是字符串，此外还有一个示例用到了 datetime.date 对象。然而，模板系统能优雅处理很多复杂的数据结构，例如列表、字典和自定义的对象。遍历 Django 模板中复杂数据结构的关键是点号（.）。

点号可以访问字典的键、属性、方法或对象的索引。最好通过几个示例说明。假如我们把一个 Python 字典传给模板。若想通过键访问那个字典中的值，要使用点号：

```
>>> from django.template import Template, Context
>>> person = {'name': 'Sally', 'age': '43'}
>>> t = Template('{{ person.name }} is {{
person.age
}} years old.')
>>> c = Context({'person': person})
>>> t.render(c)
'Sally is 43 years old.'
```

类似地，通过点号还可以访问对象的属性。例如，Python 的 datetime.date 对象有 year、month 和 day 属性，在 Django 模板中可以使用点号访问这些属性：

```
>>> from django.template import Template, Context
>>> import datetime
>>> d = datetime.date(1993, 5, 2)
>>> d.year
1993
>>> d.month
5
```

```
>>> d.day
2
>>> t = Template('The month is {{ date.month }}
and the year is {{ date.year }}.')
>>> c = Context({'date': d})
>>> t.render(c)
'The month is 5 and the year is 1993.'
```

下述示例自定义一个类，以此说明也可以通过点号访问任意对象的属性：

```
>>> from django.template import Template, Context
>>> class Person(object):
...     def __init__(self, first_name, last_name):
...         self.first_name, self.last_name =
first_name, last_name
>>> t = Template('Hello, {{ person.first_name }} {{ person.last_name }}.')
>>> c = Context({'person': Person('John', 'Smith')})
>>> t.render(c)
'Hello, John Smith.'
```

还可以通过点号引用对象的方法。例如，每个 Python 字符串都有 `upper()` 和 `isdigit()` 方法，在 Django 模板中可以使用点号句法调用它们：

```
>>> from django.template import Template, Context
>>> t = Template('{{ var }} -- {{ var.upper }} -- {{ var.isdigit }}')
>>> t.render(Context({'var': 'hello'}))
'hello -- HELLO -- False'
>>> t.render(Context({'var': '123'}))
'123 -- 123 -- True'
```

注意，方法调用中没有括号。此外，不能给方法传递变量，只能调用无需参数的方法。（本章后面将说明这里的哲学。）最后，还可以使用点号访问列表索引，例如：

```
>>> from django.template import Template, Context
>>> t = Template('Item 2 is {{ items.2 }}.')
>>> c = Context({'items': ['apples', 'bananas',
'carrots']})
>>> t.render(c)
'Item 2 is carrots.'
```

不允许使用负数索引。例如，模板变量 `{{ items.-1 }}` 会导致 `TemplateSyntaxError` 抛出。

Python 列表

提醒一下，Python 列表的索引从零开始。第一个元素的索引是 0，第二个元素的索引是 1，以此类推。

总结起来，模板系统遇到变量名中的点号时会按照下述顺序尝试查找：

- 字典查找（如 `foo["bar"]`）
- 属性查找（如 `foo.bar`）
- 方法调用（如 `foo.bar()`）

- 列表索引查找（如 foo[2]）

模板系统将使用第一个可用的类型，这是一种短路逻辑。点号查找可以嵌套多层。例如，下述示例使用的 {{ person.name.upper }}，相当于一个字典查找（person['name']）加上一个方法调用（upper()）：

```
>>> from django.template import Template, Context
>>> person = {'name': 'Sally', 'age': '43'}
>>> t = Template('{{ person.name.upper }} is {{
person.age
}} years old.')
>>> c = Context({'person': person})
>>> t.render(c)
'SALLY is 43 years old.'
```

3.3.3 方法调用的行为

方法调用与其他几种查找稍微复杂一些。下面是要知道的几件事：

- 在方法查找的过程中，如果方法抛出异常，异常会向上冒泡，除非异常有 silent_variable_failure 属性，而且值为 True。如果异常确实有 silent_variable_failure 属性，使用引擎的 string_if_invalid 配置选项（默认为一个空字符串）渲染变量。例如：

```
>>> t = Template("My name is {{ person.first_name }}.")
>>> class PersonClass3:
...     def first_name(self):
...         raise AssertionError("foo")
>>> p = PersonClass3()
>>> t.render(Context({"person": p}))
Traceback (most recent call last):
...
AssertionError: foo

>>> class SilentAssertionError(Exception):
...     silent_variable_failure = True
>>> class PersonClass4:
...     def first_name(self):
...         raise SilentAssertionError
>>> p = PersonClass4()
>>> t.render(Context({"person": p}))
'My name is .'
```

- 方法不能有必须的参数。否则，模板系统向后移动到下一种查询类型（列表索引查询）。
- 按照设计，Django 限制了在模板中可以处理的逻辑量，因此在模板中不能给方法传递参数。数据应该在视图中计算之后再传给模板显示。
- 显然，有些方法有副作用，如果允许模板系统访问这样的方法，那就愚蠢之极，甚至还可能埋下安全漏洞。
- 假如有个 BankAccount 对象，它有个 delete() 方法。如果模板中有 {{ account.delete }} 这样的内容，其中 account 是 BankAccount 对象，那么渲染模板时会把 account 删除。为了避免这种行为，在方法上设定函数属性 alters_data：

```
def delete(self):
```

```
# 删除账户
delete.alters_data = True
```

这样标记之后，模板系统不会执行方法。继续使用前面的例子。如果模板中有 `{{ account.delete }}`，而 `delete()` 方法设定了 `alters_data=True`，那么渲染模板时不会执行 `delete()` 方法，引擎会使用 `string_if_invalid` 的值替换那个变量。

注意：为 Django 模型对象动态生成的 `delete()` 和 `save()` 方法自动设定了 `alters_data = True`。

3.3.4 如何处理无效变量

一般来说，如果变量不存在，模板系统在变量处插入引擎的 `string_if_invalid` 配置选项。这个选项的默认值为一个空字符串。例如：

```
>>> from django.template import Template, Context
>>> t = Template('Your name is {{ name }}.')
>>> t.render(Context())
'Your name is .'
>>> t.render(Context({'var': 'hello'}))
'Your name is .'
>>> t.render(Context({'NAME': 'hello'}))
'Your name is .'
>>> t.render(Context({'Name': 'hello'}))
'Your name is .'
```

这一行为比抛出异常好，因为遇到人为错误时能迅速恢复。在上述示例中，所有查找都失败，因为变量名的大小写不对，或者名称错误。在现实中，如果因为模板中有小小的句法错误就导致网站不可访问，着实无法让人接受。

3.4 基本的模板标签和过滤器

前面说过，Django 的模板系统自带了一些内置的标签和过滤器。下面几节说明其中最常用的几个。

3.4.1 标签

if/else

`{% if %}` 计算变量的值，如果为真（即存在、不为空，不是假值），模板系统显示 `{% if %}` 和 `{% endif %}` 之间的内容。例如：

```
{% if today_is_weekend %}
<p>Welcome to the weekend!</p>
{% endif %}
```

`{% else %}` 标签是可选的：

```
{% if today_is_weekend %}
<p>Welcome to the weekend!</p>
{% else %}
<p>Get back to work.</p>
{% endif %}
```

`if` 标签还可以有一个或多个 `{% elif %}` 子句：

```
{% if athlete_list %}
Number of athletes: {{ athlete_list|length }}
{% elif athlete_in_locker_room_list %}
<p>Athletes should be out of the locker room soon! </p>
{% elif ...
...
{% else %}
<p>No athletes. </p>
{% endif %}
```

{% if %} 支持使用 and、or 或 not 测试多个变量，或者取反指定的变量。例如：

```
{% if athlete_list and coach_list %}
<p>Both athletes and coaches are available. </p>
{% endif %}

{% if not athlete_list %}
<p>There are no athletes. </p>
{% endif %}

{% if athlete_list or coach_list %}
<p>There are some athletes or some coaches. </p>
{% endif %}

{% if not athlete_list or coach_list %}
<p>There are no athletes or there are some coaches. </p>
{% endif %}

{% if athlete_list and not coach_list %}
<p>There are some athletes and absolutely no coaches. </p>
{% endif %}
```

在同一个标签中可以同时使用 and 和 or，此时，and 的优先级比 or 高。例如：

```
{% if athlete_list and coach_list or cheerleader_list %}
```

像下面这样解释：

```
if (athlete_list and coach_list) or cheerleader_list
```

> 注意
>
> 在 if 标签中使用括号是无效的句法。

如果需要通过括号指明优先级，应该使用嵌套的 if 标签。不支持使用括号控制操作的顺序。如果觉得有必要使用括号，可以考虑在模板外部执行逻辑，然后通过专用的模板变量传入结果。或者，直接使用嵌套的 {% if %} 标签，如下所示：

```
{% if athlete_list %}
    {% if coach_list or cheerleader_list %}
        <p>We have athletes, and either coaches or cheerleaders! </p>
    {% endif %}
{% endif %}
```

多次使用相同的逻辑运算符没问题，但是不能混用不同的运算符。例如，下述写法是有效的：

```
{% if athlete_list or coach_list or parent_list or teacher_list %}
```

每个 {% if %} 都必须有配对的 {% endif %}。否则，Django 会抛出 TemplateSyntaxError 异常。

for

{% for %} 标签用于迭代序列中的各个元素。与 Python 的 for 语句一样，句法是 for X in Y，其中 Y 是要迭代的序列，X 是单次循环中使用的变量。每次迭代时，模板系统会渲染 {% for %} 和 {% endfor %} 之间的内容。例如，可以使用下述模板显示 athlete_list 变量中的运动员：

```
<ul>
    {% for athlete in athlete_list %}
    <li>{{ athlete.name }}</li>
    {% endfor %}
</ul>
```

在标签中添加 reversed，反向迭代列表：

```
{% for athlete in athlete_list reversed %}
...
{% endfor %}
```

{% for %} 标签可以嵌套：

```
{% for athlete in athlete_list %}
<h1>{{ athlete.name }}</h1>
<ul>
    {% for sport in athlete.sports_played %}
    <li>{{ sport }}</li>
    {% endfor %}
</ul>
{% endfor %}
```

如果需要迭代由列表构成的列表，可以把每个子列表中的值拆包到独立的变量中。

比如说上下文中有一个包含 (x,y) 坐标点的列表，名为 points，可以使用下述模板输出这些坐标点：

```
{% for x, y in points %}
<p>There is a point at {{ x }},{{ y }}</p>
{% endfor %}
```

如果需要访问字典中的元素，也可以使用这个标签。如果上下文中包含一个字典 data，可以使用下述模板显示字典的键和值：

```
{% for key, value in data.items %}
{{ key }}: {{ value }}
{% endfor %}
```

通常，迭代列表之前要先检查列表的大小，如果为空，显示一些特殊的文字：

```
{% if athlete_list %}

{% for athlete in athlete_list %}
```

```
<p>{{ athlete.name }}</p>
{% endfor %}

{% else %}
<p>There are no athletes. Only computer programmers.</p>
{% endif %}
```

这种做法太常见了，因此 `for` 标签支持一个可选的 `{% empty %}` 子句，用于定义列表为空时显示的内容。下述示例与前一个等效：

```
{% for athlete in athlete_list %}
<p>{{ athlete.name }}</p>
{% empty %}
<p>There are no athletes. Only computer programmers.</p>
{% endfor %}
```

在循环结束之前，无法"跳出"。如果需要这么做，修改要迭代的变量，只包含需要迭代的值。

同样，也没有"continue"语句，不能立即返回到循环的开头。（这种设计方式背后的原因参见 3.5 节。）

在 `{% for %}` 循环内部，可以访问一个名为 `forloop` 的模板变量。这个变量有几个属性，通过它们可以获知循环进程的一些信息：

- `forloop.counter` 的值是一个整数，表示循环的次数。这个属性的值从 1 开始，因此第一次循环时，`forloop.counter` 等于 `1`。下面举个例子：

  ```
  {% for item in todo_list %}
      <p>{{ forloop.counter }}: {{ item }}</p>
  {% endfor %}
  ```

- `forloop.counter0` 与 `forloop.counter` 类似，不过是从零开始的。第一次循环时，其值为 `0`。
- `forloop.revcounter` 的值是一个整数，表示循环中剩余的元素数量。第一次循环时，`forloop.revcounter` 的值是序列中要遍历的元素总数。最后一次循环时，`forloop.revcounter` 的值为 `1`。
- `forloop.revcounter0` 与 `forloop.revcounter` 类似，不过索引是基于零的。第一次循环时，`forloop.revcounter0` 的值是序列中元素数量减去一。最后一次循环时，`forloop.revcounter0` 的值为 `0`。
- `forloop.first` 是个布尔值，第一次循环时为 `True`。需要特殊处理第一个元素时很方便：

  ```
  {% for object in objects %}
      {% if forloop.first %}
          <li class="first">
      {% else %}
          <li>
      {% endif %}

      {{ object }}
      </li>
  {% endfor %}
  ```

- `forloop.last` 是个布尔值，最后一次循环时为 `True`。经常用它在一组链接之间放置管道符号：

  ```
  {% for link in links %}
      {{ link }}{% if not forloop.last %} | {% endif %}
  {% endfor %}
  ```

上述模板代码的输出可能是：

```
Link1 | Link2 | Link3 | Link4
```

此外，还经常用它在一组单词之间放置逗号：

```
<p>Favorite places:</p>
{% for p in places %}
    {{ p }}{% if not forloop.last %}, {% endif %}
{% endfor %}
```

- 在嵌套的循环中，`forloop.parentloop` 引用父级循环的 `forloop` 对象。下面举个例子：

```
{% for country in countries %}
    <table>
        {% for city in country.city_list %}
        <tr>
            <td>Country #{{ forloop.parentloop.counter }}</td>
            <td>City #{{ forloop.counter }}</td>
            <td>{{ city }}</td>
        </tr>
        {% endfor %}
    </table>
{% endfor %}
```

`forloop` 变量只在循环内部可用。模板解析器遇到 `{% endfor %}` 时，`forloop` 随之消失。

上下文和 `forloop` 变量

在 `{% for %}` 块中，现有变量会让位，防止覆盖 `forloop` 变量。Django 把移动的上下文放到 `forloop.parentloop` 中。通常，你无须担心，但是如果有名为 `forloop` 的模板变量（不建议这么做），在 `{% for %}` 块中会重命名为 `forloop.parentloop`。

ifequal/ifnotequal

Django 模板系统不是功能全面的编程语言，因此不允许执行任意的 Python 语句（3.5 节详述）。

然而，模板经常需要比较两个值，在相等时显示一些内容。为此，Django 提供了 `{% ifequal %}` 标签。

`{% ifequal %}` 标签比较两个值，如果相等，显示 `{% ifequal %}` 和 `{% endifequal %}` 之间的内容。下述示例比较模板标签 `user` 和 `currentuser`：

```
{% ifequal user currentuser %}
    <h1>Welcome!</h1>
{% endifequal %}
```

参数可以是硬编码的字符串，使用单引号或双引号都行，因此下述代码是有效的：

```
{% ifequal section 'sitenews' %}
    <h1>Site News</h1>
{% endifequal %}

{% ifequal section "community" %}
    <h1>Community</h1>
```

```
{% endifequal %}
```

与 {% if %} 一样，{% ifequal %} 标签支持可选的 {% else %} 子句：

```
{% ifequal section 'sitenews' %}
    <h1>Site News</h1>
{% else %}
    <h1>No News Here</h1>
{% endifequal %}
```

{% ifequal %} 的参数只能是模板变量、字符串、整数和小数。下面是有效的示例：

```
{% ifequal variable 1 %}
{% ifequal variable 1.23 %}
{% ifequal variable 'foo' %}
{% ifequal variable "foo" %}
```

其他变量类型，例如 Python 字典、列表或布尔值，不能在 {% ifequal %} 中硬编码。下面是无效的示例：

```
{% ifequal variable True %}
{% ifequal variable [1, 2, 3] %}
{% ifequal variable {'key': 'value'} %}
```

如果想测试真假，应该使用 {% if %} 标签。

ifequal 标签可以替换成 if 标签和 == 运算符。

{% ifnotequal %} 的作用与 ifequal 类似，不过它测试两个参数是否不相等。ifnotequal 标签可以替换成 if 标签和 != 运算符。

注释

与 HTML 和 Python 一样，Django 模板语言支持注释。注释使用 {# #} 标明：

```
{# This is a comment #}
```

渲染模板时，不输出注释。使用这种句法编写的注释不能分成多行。这一限制有助于提升模板解析性能。

在下述模板中，渲染的结果与模板完全一样（即注释标签不会解析为注释）：

```
This is a {# this is not
a comment #}
test.
```

如果想编写多行注释，使用 {% comment %} 模板标签，如下所示：

```
{% comment %}
This is a
multi-line comment.
{% endcomment %}
```

注释标签不能嵌套。

3.4.2 过滤器

本章前面说过，模板过滤器是在显示变量之前调整变量值的简单方式。过滤器使用管道符号指定，如下所

示：

```
{{ name|lower }}
```

上述代码先通过 lower 过滤器调整 {{ name }} 变量的值——把文本转换成小写，然后再显示。过滤器可以串接，即把一个过滤器的输出传给下一个过滤器。

下述示例获取列表中的第一个元素，然后将其转换成大写：

```
{{ my_list|first|upper }}
```

有些过滤器可以接受参数。过滤器的参数放在冒号之后，始终放在双引号内。例如：

```
{{ bio|truncatewords:"30" }}
```

上述示例显示 bio 变量的前 30 个词。

下面是几个最重要的过滤器。其余的过滤器在附录 E 中说明。

- addslashes：在反斜线、单引号和双引号前面添加一个反斜线。可用于转义字符串。例如：{{ value|addslashes }}。
- date：根据参数中的格式字符串格式化 date 或 datetime 对象。例如：{{ pub_date|date:"F j, Y" }}。格式字符串在附录 E 中说明。
- length：返回值的长度。对列表来说，返回元素的数量。对字符串来说，返回字符的数量。如果变量未定义，返回 0。

3.5 理念和局限

现在，你已经大致了解了 Django Template Language（DTL），或许该说明一下背后的设计理念了。首先要知道，DTL 的局限是故意为之的。

Django 发端于在线新闻站点，其特点是大容量、变化频繁。最初设计 Django 的人对 DTL 有非常明确的理念预设。

如今，这些理念仍然是 Django 的核心。它们是：

1. 表现与逻辑分离
2. 避免重复
3. 与 HTML 解耦
4. XML 不好
5. 不要求具备设计能力
6. 透明处理空格
7. 不重造一门编程语言
8. 确保安全有保障
9. 可扩展

下面分述各点。

1. 表现与逻辑分离

模板系统用于控制表现及与其相关的逻辑，仅此而已。超出这一基本目标的功能都不应该支持。

2. 避免重复

   大多数动态网站都使用某种全站通用的设计，例如通用的页头、页脚、导航栏，等等。Django 模板系统应该为此提供便利的方式，把这些元素存储在一个位置，减少重复的代码。模板继承背后就是这个理念。

3. 与 HTML 解耦

   模板系统不应该只能输出 HTML，还要能够生成其他基于文本的格式（也就是纯文本）。

4. XML 不应该作为模板语言

   如果使用 XML 引擎解析模板，编辑模板时可能引入大量人为错误，而且处理模板有很多额外消耗。

5. 不要求具备设计能力

   模板系统不应该必须在 WYSIWYG 编辑器（如 Dreamweaver）中才能写出。这样有太多局限，句法不够灵活。

   Django 的目标是让模板编写人员能直接编辑 HTML。

6. 透明处理空格

   模板系统不应该特殊处理空格。模板中的空格就是空格，要像文本那样显示出来。不在模板标签中的空格都应该显示。

7. 不重造一门编程语言

   模板系统一定不能允许：

   - 为变量赋值
   - 编写高级的逻辑

   也就是不能重造一门编程语言。模板系统的目标是提供适量的编程功能，例如分支和循环，足够做表现相关的判断就行。

   Django 模板系统知道模板最常由设计师编写，而不是程序员，因此不要求具备 Python 知识。

8. 安全保障

   模板系统默认应该禁止包含恶意代码，例如删除数据库记录的命令。这是模板系统不允许随意使用 Python 代码的另一个原因。

9. 可扩展

   模板系统应该认识到，高级模板编写人员可能想扩展功能。这是自定义模板标签和过滤器背后的理念。

这些年我用过很多不同的模板系统，这种设计方式真心让我喜欢。DTL 以及它的设计方式是 Django 框架的主要优势之一。

在紧张的工作节奏中，设计师和程序员积极沟通，力求在截止日期之前完成工作，此时 Django 不会充当拦路虎，能让各个团队关注自己擅长做的事。

一旦你在实践中认识到这一点，很快就能理解为什么 Django 是“为快节奏完美主义者而生的框架”。

虽然如此，但是 Django 是灵活的，它不要求你必须使用 DTL。与 Web 应用的其他组件一样，模板句法具有高度主观性，程序员对此有不同的观点。单单 Python 一门语言就有几十种开源的模板语言，由此可见一斑。每出现一种新的模板语言，就说明现有模板语言不能满足开发者的需求。

Django 力求成为全栈 Web 框架，为了提升 Web 开发者的工作效率，提供了完整的组件，因此多数时候，使

用 DTL 更为便利，但这绝非严格要求。

3.6 在视图中使用模板

我们已经学习了模板系统的基础知识，下面将其运用到视图中。

回顾一下前一章在 `mysite.views` 模块中创建的 `current_datetime` 视图，如下所示：

```
from django.http import HttpResponse
import datetime

def current_datetime(request):

    now = datetime.datetime.now()
    html = "<html><body>It is now %s</body></html>" % now
    return HttpResponse(html)
```

下面修改这个视图，让它使用 Django 的模板系统。一开始你可能会想这么做：

```
from django.template import Template, Context
from django.http import HttpResponse
import datetime

def current_datetime(request):

    now = datetime.datetime.now()
    t = Template("<html><body>It is now {{ current_date }}.</body></html>")
    html = t.render(Context({'current_date': now}))
    return HttpResponse(html)
```

当然，这样也使用了模板系统，但是这没有解决我们在本章开头指出的问题，即模板仍然嵌在 Python 代码中，没有分离数据和表现。为了解决这个问题，我们要把模板放在单独的文件中，然后让视图加载。

一开始你可能会考虑把模板放在文件系统中的某个位置，然后使用 Python 内置的文件打开功能读取模板的内容。假设模板保存在 `/home/djangouser/templates/mytemplate.html` 文件中，实现这种想法的代码如下：

```
from django.template import Template, Context
from django.http import HttpResponse
import datetime

def current_datetime(request):

    now = datetime.datetime.now()
    # 使用文件系统中模板的简单方式
    # 这样做不好，因为没有考虑缺少文件的情况
    fp = open('/home/djangouser/templates/mytemplate.html')
    t = Template(fp.read())
    fp.close()

    html = t.render(Context({'current_date': now}))
    return HttpResponse(html)
```

然而，这么做不够优雅，原因如下：

- 没有处理缺少文件的情况。如果 mytemplate.html 文件不存在或不可读，open() 调用会抛出 IOError 异常。
- 模板位置是硬编码的。如果每个视图函数都这么做，要重复编写模板的位置。更别提要输入很多内容了！
- 有大量乏味的样板代码。生活如此美好，为何每次加载模板时要浪费时间编写 open()、fp.read() 和 fp.close() 调用。

为了解决这些问题，我们将使用模板加载机制，把模板放在专门的目录中。

3.7 模板加载机制

为了从文件系统中加载模板，Django 提供了便利而强大的 API，力求去掉模板加载调用和模板自身的冗余。若想使用这个模板加载 API，首先要告诉框架模板的存储位置。这个位置在设置文件中配置，即前一章介绍 ROOT_URLCONF 设置时提到的 settings.py 文件。打开 settings.py 文件，找到 TEMPLATES 设置。它的值是一个列表，分别针对各个模板引擎：

```
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [],
        'APP_DIRS': True,
        'OPTIONS': {
            # ... 一些选项 ...
        },
    },
]
```

BACKEND 的值是一个点分 Python 路径，指向实现 Django 模板后端 API 的模板引擎类。内置的后端有 django.template.backends.django.DjangoTemplates 和 django.template.backends.jinja2.Jinja2。

因为多数引擎从文件中加载模板，所以各个引擎的顶层配置包含三个通用的设置：

- DIRS 定义一个目录列表，模板引擎按顺序在里面查找模板源文件。
- APP_DIRS 设定是否在安装的应用中查找模板。按约定，APPS_DIRS 设为 True 时，DjangoTemplates 会在 INSTALLED_APPS 中的各个应用里查找名为“templates”的子目录。这样，即使 DIRS 为空，模板引擎还能查找应用模板。
- OPTIONS 是一些针对后端的设置。

同一个后端可以配置具有不同选项的多个实例，然而这并不常见。此时，要为各个引擎定义唯一的 NAME。

3.7.1 模板目录

DIRS 的默认值是一个空列表。为了告诉 Django 的模板加载机制到哪里寻找模板，选择一个选保存模板的目录，把它添加到 DIRS 中，像下面这样：

```
'DIRS': [
            '/home/html/example.com',
            '/home/html/default',
        ],
```

有几点要注意：

- 如果不是构建没有应用的极简程序，最好留空 DIRS。设置文件默认把 APP_DIRS 设为 True，因此最好在 Django 应用中放一个“templates”子目录。
- 如果想在项目根目录中放一些主模板（例如在 mysite/templates 目录中），需要像这样设定 DIRS：

```
'DIRS': [os.path.join(BASE_DIR, 'templates')],
```

- 模板目录不一定非得叫 'templates'，Django 不限制你用什么名称，但是坚守约定易于理解项目的结构。如果不想遵守这个约定，或者出于某些原因不能这么做，可以指定任何目录，只要启动 Web 服务器的用户有权读取那个目录中的模板就行。
- 在 Windows 中要加上盘符，而且要使用 Unix 风格的正斜线，而不是反斜线，如下所示：

```
'DIRS': ['C:/www/django/templates'],
```

我们还未创建 Django 应用，因此为了让下面的代码能正常运行，要把 DIRS 设为 [os.path.join(BASE_DIR,'templates')]。设定好 DIRS 之后，要修改视图代码，让它使用 Django 的模板加载机制，而不是硬编码模板的路径。回到 current_datetime 视图，把它改成下面这样：

```
from django.template.loader import get_template
from django.template import Context
from django.http import HttpResponse
import datetime

def current_datetime(request):
    now = datetime.datetime.now()
    t = get_template('current_datetime.html')
    html = t.render(Context({'current_date': now}))
    return HttpResponse(html)
```

这里，我们把手动从文件系统中加载模板的代码改成了 django.template.loader.get_template() 函数调用。这个函数的参数是模板的名称，找到模板在文件系统中的位置后，打开那个文件，编译后返回一个 Template 对象。这里使用的模板是 current_datetime.html，.html 扩展名没什么特别之处，只要你觉得合适，模板的扩展名随便用，甚至可以完全省略。

为了找到模板在文件系统中的位置，get_template() 按下列顺序查找：

- 如果 APP_DIRS 的值是 True，而且使用 DTL，在当前应用中查找“templates”目录。
- 如果在当前应用中没找到模板，get_template() 把传给它的模板名称添加到 DIRS 中的各个目录后面，按顺序在各个目录中查找。假如 DIRS 的第一个元素是 '/home/django/mysite/templates'，上述 get_template() 调用查找的模板是 /home/django/mysite/templates/current_datetime.html。
- 如果 get_template() 找不到指定名称对应的模板，抛出 TemplateDoesNotExist 异常。

为了查看模板异常是什么样子，在 Django 项目的目录中运行 python manage.py runserver，启动 Django 开发服务器。然后，在浏览器中访问激活 current_datetime 视图的页面（http://127.0.0.1:8000/time/）。假设 DEBUG 设置的值是 True，而且没有创建 current_datetime.html 模板，应该会看到 Django 错误页面，指明出现 TemplateDoesNotExist 错误（图 3-1）。

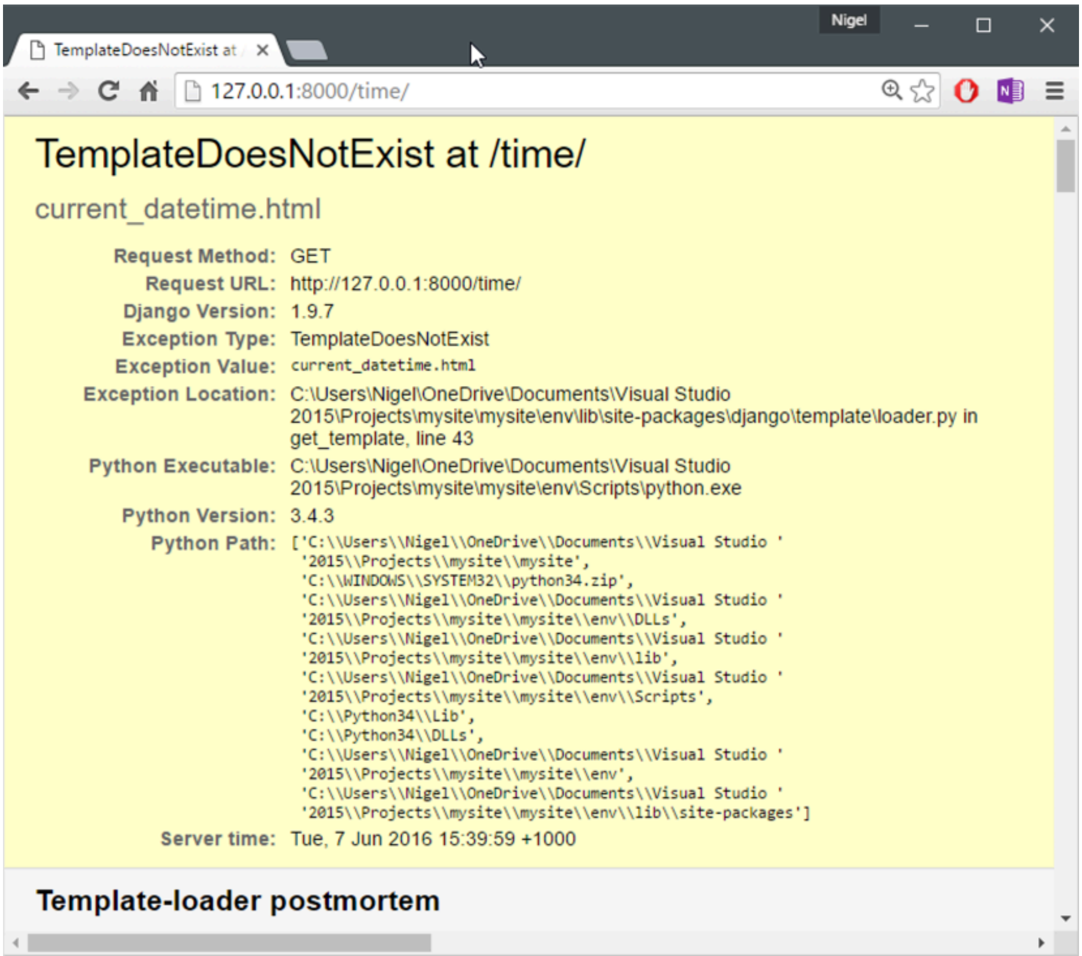


图 3-1：缺少模板的错误页面

这个错误页面与第 2 章说明的那个类似，不过多了一部分调试信息："Template-loader postmortem"。这部分信息指出 Django 尝试加载了哪些模板，以及失败的原因（例如，"File does not exist"）。调试模板加载错误时，这些信息很有价值。接下来，创建 `current_datetime.html` 文件，写入下述模板代码：

```
It is now {{ current_date }}.
```

把这个文件保存到 `mysite/templates` 目录中（如果没有"templates"目录，创建一个）。然后在 Web 浏览器中刷新页面，此时应该能看到正确渲染的页面。

3.8 render()

至此，我们知道如何加载模板、填充上下文、返回 `HttpResponse` 对象了，这么做的结果是渲染一个模板。随后，我们优化了视图，使用 `get_template()` 替换硬编码的模板路径。我这么做是为了让你了解 Django 加载及在浏览器中渲染模板的过程。

其实，Django 为此提供了更为简单的方式。Django 的开发者意识到这是一个常见的任务，因此需要一种简单的方式，只需一行代码就能做到。这个简单方式是 `django.shortcuts` 模块中名为 `render()` 的函数。

多数时候人们都使用 `render()`，而不自己动手加载模板、创建 `Context` 和 `HttpResponse` 对象——除非雇主以代码行数评判你的工作。

下面是使用 render() 重写的 current_datetime 视图：

```
from django.shortcuts import render
import datetime

def current_datetime(request):
    now = datetime.datetime.now()
    return render(request, 'current_datetime.html', {'current_date': now})
```

差别多么明显啊！下面详细说明这次改动：

- 不用再导入 get_template、Template、Context 或 HttpResponse 了，而要导入 django.shortcuts.render。import datetime 不变。
- 在 current_datetime 函数中，仍然要计算 now，不过加载模板、创建上下文、渲染模板和创建 HttpResponse 对象全由 render() 调用代替了。render() 的返回值是一个 HttpResponse 对象，因此在视图中可以直接返回那个值。

render() 的第一个参数是请求对象，第二个参数是模板名称，第三个单数可选，是一个字段，用于创建传给模板的上下文。如果不指定第三个参数，render() 使用一个空字典。

3.9 模板子目录

如果把所有模板存放在一个目录中，很快就会变得不灵便。你可能想把模板存储在模板目录的子目录里，这么做很好。

其实，我也推荐这么做，因为有些高级的 Django 功能（例如第 10 章将说明的通用视图系统）默认预期这种模板布局方式。

把模板放到模板目录的子目录中不是难事。调用 get_template() 时，只需在模板名称前面加上子目录的名称和一条斜线，如下所示：

```
t = get_template('dateapp/current_datetime.html')
```

render() 是对 get_template() 的简单包装，因此在 render() 的第二个参数中也可以这么做，如下所示：

```
return render(request, 'dateapp/current_datetime.html', {'current_date': now})
```

子目录的层级深度没有限制，可以根据需要尽情使用。

> **提示**
>
> Windows 用户注意，要使用正斜线，而非反斜线。get_template() 要求 Unix 风格的文件名。

3.10 include 模板标签

说完模板加载机制之后，可以介绍利用这一机制的一个内置模板标签了：{% include %}。

这个标签的作用是引入另一个模板的内容。它的参数是要引入的模板的名称，可以是变量，也可以是硬编码的字符串（放在引号里，单双引号都行）。

只要想在多个模板中使用相同的代码，就可以考虑使用 {% include %}，去除重复。下面两个示例引入

nav.html 模板的内容。二者的作用相同，目的是说明既可以使用单引号，也可以使用双引号。

```
{% include 'nav.html' %}
{% include "nav.html" %}
```

下述示例引入 includes/nav.html 模板的内容：

```
{% include 'includes/nav.html' %}
```

下述示例引入的模板名称由变量 template_name 指定：

```
{% include template_name %}
```

与 get_template() 一样，include 标签的参数指定的模板在当前 Django 应用的“templates”目录中（APPS_DIR 为 True 时），或者在 DIRS 设置的目录中。引入的模板在引入它的模板的上下文中执行。

来看下面两个模板：

```
# mypage.html

<html>
<body>
    {% include "includes/nav.html" %}
    <h1>{{ title }}</h1>
</body>
</html>

# includes/nav.html

<div id="nav">
    You are in: {{ current_section }}
</div>
```

如果渲染 mypage.html 时使用的上下文中有 current_section，那么它可以作为变量在引入的模板中使用——这与我们预期的一样。

如果 {% include %} 标签的参数指定的模板不存在，Django 会做下面两件事中的一件：

- DEBUG 为 True 时，渲染 Django 错误页面，显示 TemplateDoesNotExist 异常。
- DEBUG 为 False 时，静默，那个标签的位置什么也不显示。

提示

引入的模板之间没有共享的状态，每次引入都是完全独立的渲染过程。块在引入之前运行。这意味着，从另一个模板中引入的块已经执行并渲染了，不能在引入它的模板中覆盖。

3.11 模板继承

目前，我们举的示例都是简短的 HTML 片段，但是在真实世界中，你将使用 Django 的模板系统创建整个 HTML 页面。这就引出 Web 开发常见的一个问题：在整个网站中，如何减少通用页面区域（如全站导航）的重复和冗余？

这个问题一种典型的解决方法是使用服务器端引入，即在 HTML 页面中使用指令“引入”另一个网页。其实，Django 支持这个方法，即使用前面说过的 `{% include %}` 模板标签。

但是，在 Django 中解决这个问题更好的方法是使用模板继承，这样更优雅。大致来讲，模板继承是指创建一个基底“骨架”模板，包含网站的所有通用部分，并且定义一些“块”，让子模板覆盖。下面举个例子。编辑 `current_datetime.html` 文件，为 `current_datetime` 视图创建更为完整的模板：

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN">

<html lang="en">

<head>
    <title>The current time</title>
</head>
<body>
    <h1>My helpful timestamp site</h1>
    <p>It is now {{ current_date }}.</p>

    <hr>
    <p>Thanks for visiting my site.</p>
</body>
</html>
```

这看起来没什么问题，但是如果要为另一个视图创建模板呢？比如第 2 章编写的 `hours_ahead` 视图。如果还想编写一个精美有效的完整 HTML 模板，要这么做：

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN">

<html lang="en">

<head>
    <title>Future time</title>
</head>
<body>
    <h1>My helpful timestamp site</h1>
    <p>
        In {{ hour_offset }} hour(s), it will be {{ next_time }}.
    </p>

    <hr>
    <p>Thanks for visiting my site.</p>
</body>
</html>
```

显然，刚刚重复编写了大量 HTML。想象一下，典型的网站包含导航栏、几个样式表，可能还有一些 JavaScript，这得在各个模板中编写多少重复的 HTML 啊！

服务器端引入方案解决这个问题的方法是把模板中通用的部分提取出来，保存到单独的模板中，然后再引入各个模板。或许可以把模板顶部那部分保存在 `header.html` 文件中：

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN">

<html lang="en">
```

```
<head>
```

把底部保存在 footer.html 文件中：

```
    <hr>
    <p>Thanks for visiting my site.</p>
    </body>
</html>
```

采用这种方式，页头和页脚处理起来很容易，真正难处理的是中间部分。这里，两个页面都有同一个标头（“My helpful timestamp site”），但是不能把它放在 header.html 文件中，因为两个页面的标题不同。如果把那个标头放在页头中，标题也在其中，这样就不能在各个页面中定制标题了。

Django 的模板继承系统能解决这样的问题。你可以把它理解为服务器端引入的相反版本。我们不再定义通用的片段，而是定义有区别的片段。

首先，要定义一个基模板，即一个骨架，供子模板填充。下面是针对前述示例的基模板：

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN">

<html lang="en">

<head>
    <title>{% block title %}{% endblock %}</title>
</head>
<body>
    <h1>My helpful timestamp site</h1>
    {% block content %}{% endblock %}

    {% block footer %}
    <hr>
    <p>Thanks for visiting my site.</p>
    {% endblock %}
</body>
</html>
```

我们把这个模板命名为 base.html，它定义了一个简单的 HTML 文档骨架，供网站中的所有页面使用。

子模板可以覆盖块的内容、向块中添加内容，或者原封不动。（如果你一直跟着做，请把这个文件保存到模板目录中，命名为 base.html。）

这里用到一个你没见过的模板标签：{% block %}。它的作用很简单，告诉模板引擎，子模板可以覆盖这部分内容。

创建好基模板之后，修改现有的 current_datetime.html 模板，让它使用基模板：

```
{% extends "base.html" %}

{% block title %}The current time{% endblock %}

{% block content %}
    <p>It is now {{ current_date }}.</p>
{% endblock %}
```

此外，再为第 3 章编写的 hours_ahead 视图创建一个模板吧。(把 hours_ahead 视图中硬编码的 HTML 改为使

用模板系统这个任务留给你去做。)下面是这个模板的代码：

```
{% extends "base.html" %}

{% block title %}Future time{% endblock %}

{% block content %}
<p>
    In {{ hour_offset }} hour(s), it will be {{ next_time }}.
</p>
{% endblock %}
```

这样是不是很棒？各个模板只包含自己专用的代码，没有冗余。如果想修改全站的设计，只需修改 base.html，其他所有模板都能立即体现出变化。

下面说说原理。模板引擎加载 current_datetime.html 模板时，发现有 {% extends %} 标签，意识到这是一个子模板，因此立即加载父模板，即这里的 base.html。

加载父模板时，模板引擎发现 base.html 中有三个 {% block %} 标签，然后使用子模板中的内容替换。因此，将使用 {% block title %} 中定义的标题和 {% block content %} 中定义的内容。

注意，这个子模板没有定义 footer 块，因此模板系统使用父模板中的内容。父模板中的 {% block %} 块总是后备内容。

继承不影响模板的上下文。也就是说，继承树中的任何模板都能访问上下文中的每一个模板变量。根据需要，继承层级的深度不限。继承经常使用下述三层结构：

1. 创建 base.html 模板，定义网站的整体外观。这个模板的内容很少变化。
2. 为网站中的各个“区域”创建 base_SECTION.html 模板（如 base_photos.html 和 base_forum.html）。这些模板扩展 base.html，包含各区域专属的样式和设计。
3. 为各种页面创建单独的模板，例如论坛页面或相册。这些模板扩展相应的区域模板。

这种方式能最好地复用代码，而且便于为共享的区域添加内容，例如同一个区域通用的导航。

下面是使用模板继承的一些指导方针：

- 如果模板中有 {% extends %}，必须是模板中的第一个标签。否则，模板继承不起作用。
- 一般来说，基模板中的 {% block %} 标签越多越好。记住，子模板无需定义父模板中的全部块，因此可以为一些块定义合理的默认内容，只在子模板中覆盖需要的块。钩子多总是好的。
- 如果发现要在多个模板中重复编写相同的代码，或许说明应该把那些代码移到父模板中的一个 {% block %} 标签里。
- 如果需要从父模板中的块里获取内容，使用 {{ block.super }}，这是一个“魔法”变量，提供父模板中渲染后的文本。向块中添加内容，而不是完全覆盖时就可以这么做。
- 在同一个模板中不能为多个 {% block %} 标签定义相同的名称。之所以有这个限制，是因为 block 标签是双向的。即，block 标签不仅标识供填充的空位，还用于定义填充父模板中空位的内容。如果一个模板中有两个同名的块，那么父模板就不知道使用哪个块里的内容。
- 传给 {% extends %} 的模板名称使用与 get_template() 相同的方法加载。即，模板在 DIRS 设置定义的目录中，或者在当前 Django 应用的“templates”目录里。
- 多数情况下，{% extends %} 的参数是字符串，不过如果直到运行时才知道父模板的名称，也可以用变量。通过这一点可以做些动态判断。

3.12 接下来

现在，你掌握了 Django 模板系统的基础知识，接下来呢？多数现代的网站是数据库驱动的，即网站的内容存储在关系数据库中。这把数据和逻辑明确区分开了（与视图和模板把逻辑和表现分开是一个道理）。下一章介绍 Django 提供的与数据库交互的工具。

第 4 章 Django 模型

第 2 章涵盖了使用 Django 构建动态网站的基础，即设置视图和 URL 配置。如前所述，视图负责执行逻辑，然后返回响应。在所举的示例中，有一个的逻辑是计算当前日期和时间。

对现代的 Web 应用程序而言，视图逻辑经常需要与数据库交互。在数据库驱动型网站中，网站连接数据库服务器，从中检索数据，然后在网页中把数据显示出来。此外，可能还会提供让访客自行填充数据库的方式。

很多复杂的网站都兼具这两种操作。亚马逊网站就是数据库驱动型网站，每个商品页面其实都会查询亚马逊的商品数据库，然后把数据以 HTML 格式显示出来；顾客发表评论时，把评论插入相应的数据库表中。

Django 非常适合构建数据库驱动型网站，它提供了简单而强大的工具，易于使用 Python 执行数据库查询。本章就说明这个功能，即 Django 的数据库层。

提示

严格来说，使用 Django 的数据库层不必知道基本的关系数据库理论和 SQL，但是强烈建议你掌握一些这方面的知识。本书不会涉及这些知识，但是就算你是数据库新手，也可以继续往下读，说不定在这个过程中你能掌握一些呢。

4.1 在视图中执行数据库查询的“愚蠢”方式

第 2 章详述了在视图中生成输出的“愚蠢”方式（直接在视图中硬编码文本），同样，在视图中从数据库里检索数据也有“愚蠢”方式。这种方式很简单：使用现有的 Python 库执行 SQL 查询，然后处理结果。在下面这个示例中，我们使用 `MySQLdb` 库连接一个 MySQL 数据库，检索一些记录，提供给模板，在网页中显示：

```
from django.shortcuts import render
import MySQLdb

def book_list(request):
    db = MySQLdb.connect(user='me', db='mydb',  passwd='secret', host='localhost')
    cursor = db.cursor()
    cursor.execute('SELECT name FROM books ORDER BY name')
    names = [row[0] for row in cursor.fetchall()]
    db.close()
    return render(request, 'book_list.html', {'names': names})
```

这么做是可以，但是很快就会出现一些问题：

- 数据库连接参数是硬编码的。理想的做法是，把这些参数存储在 Django 配置中。
- 要编写相当多的样板代码：建立连接、创建游标、执行语句、关闭连接。理想情况下，我们只应该指定想要什么结果。
- 与 MySQL 耦合。如果以后想从 MySQL 转到 PostgreSQL，要重新编写大量代码。理想情况下，数据库服务器应该有一层抽象，这样在一处修改就能更换。（如果构建的是开源 Django 应用程序，希望有更多的人使用，这个特性尤其有用。）

正如你期待的那样，Django 的数据库层解决了这些问题。

4.2 配置数据库

了解基本原理之后，下面来讨论 Django 的数据库层。首先，看一下创建应用程序时在 `settings.py` 文件中添加的初始配置：

```
# Database
# ...
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': os.path.join(BASE_DIR, 'db.sqlite3'),
    }
}
```

默认的设置非常简单，下面说明各个设置：

- `ENGINE` 告诉 Django 使用哪个数据库引擎。本书的示例使用 SQLite，因此不用改，继续使用默认的 `django.db.backends.sqlite3`。
- `NAME` 告诉 Django 数据库的名称。例如：`'NAME': 'mydb',`。

因为我们使用的是 SQLite，`startproject` 命令为我们填上了数据库文件的完整文件系统路径。

这是默认设置，对这本书中的代码来说，无需修改。这里讲解这些设置，是为了让你了解在 Django 中配置数据库是多么简单。第 21 章将详细说明如何配置 Django 支持的各种数据库。

4.3 第一个应用

确认可以连接数据库之后，接下来我们将创建一个 Django 应用——一系列 Django 代码，包含模型和视图，打包在一个独立的 Python 包里，表示一个完整的 Django 应用程序。这里有必要澄清一下术语，因为初学者容易弄混。我们在第 1 章创建了一个项目，那么项目与应用之间有什么区别呢？区别是一个是配置，一个是代码。

- 一个项目是一系列 Django 应用的实例，外加那些应用的配置。严格来说，一个项目唯一需要的是一个设定文件，定义数据库连接信息、安装的应用列表、`DIRS`，等等。
- 一个应用是一系列便携的 Django 功能，通常包含模型和视图。打包在一个 Python 包里。Django 自带了一些应用，例如管理后台。这些应用的独特之处是便携，可以在多个项目中复用。

你编写的代码在这二者之间游走，没有严格的界限。如果构建的是简单的网站，可能只会使用一个应用；如果构建复杂的网站，有几个不相关的部分，如电商系统和留言板，可能想把各部分放在单独的应用中，这样以后可以复用。

其实，并非一定要创建应用，本书目前所举的视图函数示例都没这么做，而是创建一个名为 `views.py` 的文件，在里面编写视图函数，然后把 URL 配置指向那些函数。没有任何“应用”是必须的。

然而，在应用方面有个严守的约定：如果使用 Django 的数据库层（模型），必须创建 Django 应用。模型必须保存在应用中。因此，编写模型之前，要新建一个应用。

在 `mysite` 项目目录（`manage.py` 文件所在的目录，不是 `mysite` 应用目录）里输入下述命令创建 books 应用：

```
python manage.py startapp books
```

这个命令没有输出，不过却会在 `mysite` 目录中创建 `books` 子目录。我们来看一下那个目录的内容：

```
books/
    /migrations
    __init__.py
    admin.py
    models.py
    tests.py
    views.py
```

这个应用的模型和视图就保存在这些文件中。在你喜欢的文本编辑器中看一下 `models.py` 和 `views.py` 文件的内容。这两个文件除了一个导入语句和一行注释之外，没有其他内容。这是 Django 应用的崭新起点。

4.4 使用 Python 定义模型

第 1 章说过，“MTV”中的“M”表示“Model”（模型）。Django 模型是使用 Python 代码对数据库中数据的描述，是数据的结构，等效于 SQL 中的 `CREATE TABLE` 语句，不过是用 Python 代码而非 SQL 表述的，而且不仅包含数据库列的定义。

Django 通过模型在背后执行 SQL，返回便利的 Python 数据结构，表示数据库表中的行。Django 还通过模型表示 SQL 无法处理的高层级概念。

如果熟悉数据库，你的第一反应可能是：“使用 Python 代替 SQL 定义数据模型是不是多此一举？”Django 之所以这么做有几个原因：

- *内省*（introspection）有开销，而且不完美。为了提供便利的数据访问 API，Django 需要以某种方式知晓数据库布局，而这一需求有两种实现方式。第一种是使用 Python 明确描述数据，第二种是在运行时内省数据库，推知数据模型。第二种方式在一个地方存储表的元数据，看似更简单，其实会导致几个问题。首先，运行时内省数据库肯定有消耗。如果每次执行请求，或者只是初始化 Web 服务器都要内省数据库，那带来的消耗是无法接受的。（有些人觉得那点消耗不算事，然而 Django 的开发者可是在想方设法努力降低框架的消耗。）其次，有些数据库，尤其是旧版 MySQL，存储的元数据不足以完成内省。
- Python 编写起来让人心情舒畅，而且使用 Python 编写所有代码无需频繁让大脑切换情境。

  在一个编程环境（思维）中待久了，有助于提升效率。在 SQL 和 Python 之间换来换去容易打断状态。
- 把数据模型保存在代码中比保存在数据库中易于做版本控制，易于跟踪数据布局的变化。
- SQL 对数据布局的元数据只有部分支持。例如，多数数据库系统没有提供专门表示电子邮件地址或 URL 的数据类型。而 Django 模型有。高层级的数据结构有助于提升效率，让代码更便于复用。
- 不同数据库平台使用的 SQL 不一致。

  分发 Web 应用程序时，更务实的做法是分发一个描述数据布局的 Python 模块，而不是分别针对 MySQL、PostgreSQL 和 SQLite 的 `CREATE TABLE` 语句。

不过，这种方式有个缺点：模型的 Python 代码可能与数据库的真正结构脱节。如果修改了 Django 模型，还要在数据库中做相同的改动，让数据库与模型保持一致。本章后面讨论*迁移*（migration）时会说明如何处理这个问题。

最后，要知道 Django 提供了一个实用程序，可以内省现有的数据库，生成模型。有了这个程序，可以快速让旧代码运行起来。详情参见第 21 章。

4.4.1 第一个模型

为了给本章和下一章提供一个连贯的示例，我将实现一个基本的“图书-作者-出版社”数据布局。我之所以以此为例，是因为图书、作者和出版社之间的关系是大家熟知的，而且 SQL 入门书经常使用这样的数据布局。你现在阅读的这本书就是由几位作者撰写，再由出版社出版的。

下面对一些概念、字段和关系做个说明：

- 作者有名字、姓和电子邮件地址。
- 出版社有名称、街道地址、所在城市、州（省）、国家和网站。
- 书有书名和出版日期，还有一位或多位作者（与作者是多对多关系），以及一个出版社〔出版社（外键）与书是一对多关系〕。

在 Django 中使用这个数据库布局的第一步是使用 Python 代码把它表示出来。在 `startapp` 命令创建的 `models.py` 文件中输入下述内容：

```
from django.db import models

class Publisher(models.Model):
    name = models.CharField(max_length=30)
    address = models.CharField(max_length=50)
    city = models.CharField(max_length=60)
    state_province = models.CharField(max_length=30)
    country = models.CharField(max_length=50)
    website = models.URLField()

class Author(models.Model):
    first_name = models.CharField(max_length=30)
    last_name = models.CharField(max_length=40)
    email = models.EmailField()

class Book(models.Model):
    title = models.CharField(max_length=100)
    authors = models.ManyToManyField(Author)
    publisher = models.ForeignKey(Publisher)
    publication_date = models.DateField()
```

下面简单说明这段代码，介绍一些基本知识。首先，每个模型使用一个 Python 类表示，而且是 `django.db.models.Model` 的子类。父类 `Model` 包含与数据库交互所需的全部机制，而且只让模型以简洁明了的句法定义字段。

你可能不信，让 Django 具有基本的数据访问能力只需编写这些代码。一般，一个模型对应于一个数据库表，模型中的各个属性分别对应于数据库表中的一列。属性的名称对应于列的名称，字段的类型（如 `CharField`）对应于数据库列的类型（如 `varchar`）。例如，`Publisher` 模型等效于下述表（假定使用 PostgreSQL 的 `CREATE TABLE` 句法）：

```
CREATE TABLE "books_publisher" (
    "id" serial NOT NULL PRIMARY KEY,
    "name" varchar(30) NOT NULL,
    "address" varchar(50) NOT NULL,
    "city" varchar(60) NOT NULL,
    "state_province" varchar(30) NOT NULL,
```

```
    "country" varchar(50) NOT NULL,
    "website" varchar(200) NOT NULL
);
```

其实，稍后你将看到，Django 能自动生成 CREATE TABLE 语句。一个类对应一个数据库表在一种情况下有例外：多对多关系。在上述示例模型中，Book 有一个 ManyToManyField 属性，即 authors。这指明一本书有一位或多位作者，但是 Book 数据库表中没有 authors 列。此时，Django 会创建一个额外的表，一个多对多联结表（join table），处理书与作者之间的对应关系。

完整的字段类型和模型句法选项参见附录 B。最后请注意，我们没有在任何一个模型中定义主键。如果没有明确定义，Django 会自动为每个模型定义一个自增量整数主键字段，名为 id。每个 Django 模型都必须有一个主键列。

4.4.2 安装模型

编写好代码之后，要在数据库中创建表。为此，第一步是在 Django 项目中激活那些模型。激活的方法是把 books 应用添加到设置文件中"安装的应用"列表中。打开 settings.py 文件，找到 INSTALLED_APPS 设置。它的作用是告诉 Django，当前项目激活了哪些应用。默认情况下，它的值如下：

```
INSTALLED_APPS = (
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
)
```

为了注册我们开发的"books"应用，要把 'books'（指代我们正在开发的"books"应用）添加到 INSTALLED_APPS 中，得到下述设置：

```
INSTALLED_APPS = (
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'books',
)
```

INSTALLED_APPS 中的各个应用通过完整的 Python 路径表示，即点分包路径，直到应用所在的包。在设置文件中激活 Django 应用之后，可以在数据库中创建表了。首先，运行下述命令，验证模型：

```
python manage.py check
```

check 命令运行 Django 系统检查框架，即验证 Django 项目的一系列静态检查。如果一切正常，你将看到这个消息：System check identified no issues (0 silenced)。如若不然，请确保你输入的模型代码是正确的。错误消息应该会告诉你代码哪里出错了。只要觉得模型有问题，就可以运行 python manage.py check，它能捕获全部常见的模型问题。

确认模型有效之后，运行下述命令，告诉 Django 你对模型做了修改（这里是新建了模型）：

```
python manage.py makemigrations books
```

你应该会看到类似下面的输出：

```
Migrations for 'books':
  0001_initial.py:
    - Create model Author
    - Create model Book
    - Create model Publisher
    - Add field publisher to book
```

Django 把对模型（也就是数据库模式）的改动存储在迁移中，迁移就是磁盘中的文件。运行上述命令后，books 应用的 migrations 文件夹里会出现一个名为 0001_initial.py 的文件。migrate 命令会查看最新的迁移文件，自动更新数据库模式；不过，我们先来看看将运行的 SQL。sqlmigrate 命令的参数是迁移名称，输出的结果是对应的 SQL：

```
python manage.py sqlmigrate books 0001
```

你应该看到类似下面的输出（为了便于阅读，重新做了排版）：

```sql
BEGIN;

CREATE TABLE "books_author" (
    "id" integer NOT NULL PRIMARY KEY AUTOINCREMENT,
    "first_name" varchar(30) NOT NULL,
    "last_name" varchar(40) NOT NULL,
    "email" varchar(254) NOT NULL
);
CREATE TABLE "books_book" (
    "id" integer NOT NULL PRIMARY KEY AUTOINCREMENT,
    "title" varchar(100) NOT NULL,
    "publication_date" date NOT NULL
);
CREATE TABLE "books_book_authors" (
    "id" integer NOT NULL PRIMARY KEY AUTOINCREMENT,
    "book_id" integer NOT NULL REFERENCES "books_book" ("id"),
    "author_id" integer NOT NULL REFERENCES "books_author" ("id"),
    UNIQUE ("book_id", "author_id")
);
CREATE TABLE "books_publisher" (
    "id" integer NOT NULL PRIMARY KEY AUTOINCREMENT,
    "name" varchar(30) NOT NULL,
    "address" varchar(50) NOT NULL,
    "city" varchar(60) NOT NULL,
    "state_province" varchar(30) NOT NULL,
    "country" varchar(50) NOT NULL,
    "website" varchar(200) NOT NULL
);
CREATE TABLE "books_book__new" (
    "id" integer NOT NULL PRIMARY KEY AUTOINCREMENT,
    "title" varchar(100) NOT NULL,
    "publication_date" date NOT NULL,
    "publisher_id" integer NOT NULL REFERENCES
    "books_publisher" ("id")
);
```

```
INSERT INTO "books_book__new" ("id", "publisher_id", "title",
"publication_date") SELECT "id", NULL, "title", "publication_date" FROM
"books_book";

DROP TABLE "books_book";

ALTER TABLE "books_book__new" RENAME TO "books_book";

CREATE INDEX "books_book_2604cbea" ON "books_book" ("publisher_id");

COMMIT;
```

注意以下几点：

- 自动生成的表名结合应用的名称（books）和模型名的小写形式（publisher、book 和 author）。这个行为可以覆盖，详情参阅附录 B。
- 前面说过，Django 会自动为各个表添加主键，即 id 字段。这个行为可以覆盖。按约定，Django 在外键字段的名称后面加上 "_id"。你可能猜到了，这个行为也可以覆盖。
- 外键关系通过 REFERENCES 语句指明。

CREATE TABLE 语句会针对所用的数据库调整，因此能自动处理数据库专有的字段类型，例如 auto_increment（MySQL）、serial（PostgreSQL）或 integer primary key（SQLite）。列名的引号也是如此（例如使用双引号或单引号）。上述示例输出用的是 PostgreSQL 句法。

sqlmigrate 命令并不创建表，其实它根本不接触数据库，而是在屏幕上输出 Django 将执行的 SQL。如果愿意，可以把输出的 SQL 复制粘贴到数据库客户端里，然而，Django 为提交 SQL 提供了更为简单的方式——migrate 命令：

```
python manage.py migrate
```

运行这个命令后将看到类似下面的输出：

```
Operations to perform:
  Apply all migrations: books
Running migrations:
  Rendering model states... DONE

  # ...

  Applying books.0001_initial... OK

  # ...
```

那些额外的输出（注释掉的部分）是首次运行迁移时 Django 创建的系统表，是 Django 内置应用所需的。迁移是 Django 把模型改动（添加字段、删除模型，等等）应用到数据库模式的方式。大多数情况下，迁移能自动完成工作，然而也有一些不足。关于迁移的更多信息，参阅第 21 章。

4.5 基本的数据访问

创建模型之后，Django 自动提供了操作模型的高层 Python API。运行 python manage.py shell，输入下述代码试试：

```
>>> from books.models import Publisher
>>> p1 = Publisher(name='Apress', address='2855 Telegraph Avenue',
...     city='Berkeley', state_province='CA', country='U.S.A.',
...     website='http://www.apress.com/')
>>> p1.save()
>>> p2 = Publisher(name="O'Reilly", address='10 Fawcett St.',
...     city='Cambridge', state_province='MA', country='U.S.A.',
...     website='http://www.oreilly.com/')
>>> p2.save()
>>> publisher_list = Publisher.objects.all()
>>> publisher_list
[<Publisher: Publisher object>, <Publisher: Publisher object>]
```

这么几行代码完成的工作很多。下面着重说明几点：

- 首先，导入 Publisher 模型类，以便与保存出版社的数据库表交互。
- 提供各个字段的值，name、address，等等，实例化一个 Publisher 对象。
- 为了把对象保存到数据库中，调用 save() 方法。Django 在背后执行 SQL INSERT 语句。
- 为了从数据库中检索出版社，使用 Publisher.objects 属性，你可以把它的值理解为全部出版社。使用 Publisher.objects.all() 获取数据库中的所有 Publisher 对象。Django 在背后执行 SQL SELECT 语句。

有一件事在上述示例中可能没有明确体现出来，要提一下。使用 Django 模型 API 创建的对象不会自动保存，只能自己动手调用 save() 方法：

```
p1 = Publisher(...)
# 此时，p1 尚未保存到数据库中！
p1.save()
# 现在保存了。
```

如果想在一步中创建对象并保存到数据库中，使用 objects.create() 方法。下述示例与前述示例等效：

```
>>> p1 = Publisher.objects.create(name='Apress',
...     address='2855 Telegraph Avenue',
...     city='Berkeley', state_province='CA', country='U.S.A.',
...     website='http://www.apress.com/')
>>> p2 = Publisher.objects.create(name="O'Reilly",
...     address='10 Fawcett St.', city='Cambridge',
...     state_province='MA', country='U.S.A.',
...     website='http://www.oreilly.com/')
>>> publisher_list = Publisher.objects.all()
>>> publisher_list
[<Publisher: Publisher object>, <Publisher: Publisher object>]
```

当然，使用 Django 数据库 API 能执行相当多的操作，不过我们先来处理一件让人烦恼的小事。

4.5.1 添加模型的字符串表示形式

打印出版社列表时，得到的是下述没有什么用的输出，不易区分各个 Publisher 对象：

```
[<Publisher: Publisher object>, <Publisher: Publisher object>]
```

这个问题易于修正，为 Publisher 类添加一个名为 __str__() 的方法即可。__str__() 方法的作用是告诉 Python 如何以人类可读的形式显示对象。我们为那三个模型添加 __str__() 方法，看看效果：

```
from django.db import models

class Publisher(models.Model):
    name = models.CharField(max_length=30)
    address = models.CharField(max_length=50)
    city = models.CharField(max_length=60)
    state_province = models.CharField(max_length=30)
    country = models.CharField(max_length=50)
    website = models.URLField()

    def __str__(self):
        return self.name

class Author(models.Model):
    first_name = models.CharField(max_length=30)
    last_name = models.CharField(max_length=40)
    email = models.EmailField()

    def __str__(self):
        return u'%s %s' % (self.first_name, self.last_name)

class Book(models.Model):
    title = models.CharField(max_length=100)
    authors = models.ManyToManyField(Author)
    publisher = models.ForeignKey(Publisher)
    publication_date = models.DateField()

    def __str__(self):
        return self.title
```

可以看出，为了返回对象的表示形式，__str__() 方法可以做任何需要做的事情。这里，Publisher 和 Book 类的 __str__() 方法只是分别返回对象的名称和标题，而 Author 类的 __str__() 方法稍微复杂一些，把 first_name 和 last_name 字段放在一起，以一个空格分开。对 __str__() 方法的唯一要求是返回一个字符串对象。如若不然，而是返回整数，Python 会抛出 TypeError 异常，显示下述消息：

```
TypeError: __str__ returned non-string (type int).
```

为了让增加的 __str__() 方法起作用，退出 Python shell，再运行 python manage.py shell 命令进入。（这是让代码改动起作用的最简单方式。）现在，Publisher 对象列表容易理解多了：

```
>>> from books.models import Publisher
>>> publisher_list = Publisher.objects.all()
>>> publisher_list
[<Publisher: Apress>, <Publisher: O'Reilly>]
```

每个模型都应该定义 __str__() 方法，这样做不仅是为了你自己在使用交互式解释器时提供便利，还因为 Django 在显示对象的多个地方需要用到 __str__() 方法的输出。最后注意一点，__str__() 是为模型添加行为的好例子。Django 模型不仅仅用于描述对象的数据库表布局，还用于描述对象知道如何处理的功能。__str__() 就是其中一例，其作用是让模型知道如何显示自身。

4.5.2 插入和更新数据

我们已经见过如何在数据库中插入一行，先使用关键字参数创建模型的实例，如下所示：

```
>>> p = Publisher(name='Apress',
...         address='2855 Telegraph Ave.',
...         city='Berkeley',
...         state_province='CA',
...         country='U.S.A.',
...         website='http://www.apress.com/')
```

前面说过，实例化模型类不会接触数据库。为了把记录保存到数据库中，要像下面这样调用 save() 方法：

```
>>> p.save()
```

这一步基本上相当于下述 SQL 语句：

```
INSERT INTO books_publisher
    (name, address, city, state_province, country, website)
VALUES
    ('Apress', '2855 Telegraph Ave.', 'Berkeley', 'CA',
     'U.S.A.', 'http://www.apress.com/');
```

Publisher 模型有个自增量主键 id，因此调用 save() 方法后还会做一件事：计算记录的主键值，把它赋值给实例的 id 属性。

```
>>> p.id
52    # 你得到的结果可能不同
```

后续再调用 save() 将就地保存记录，而不新建记录（即，执行 SQL UPDATE 语句，而非 INSERT 语句）：

```
>>> p.name = 'Apress Publishing'
>>> p.save()
```

上述 save() 调用基本上相当于执行下述 SQL 语句：

```
UPDATE books_publisher SET
    name = 'Apress Publishing',
    address = '2855 Telegraph Ave.',
    city = 'Berkeley',
    state_province = 'CA',
    country = 'U.S.A.',
    website = 'http://www.apress.com'
WHERE id = 52;
```

是的，注意所有字段都将更新，而不是只更新有变化的字段。这在某些逻辑中可能导致条件竞争。执行下述查询（稍有不同）的方法参见 4.5.9 节：

```
UPDATE books_publisher SET
    name = 'Apress Publishing'
WHERE id=52;
```

4.5.3 选择对象

知道如何创建和更新数据库记录是基本的，但是你构建的 Web 应用程序可能更多的是查询现有对象，而非新

建。我们已经见过检索指定模型中每个记录的一种方式：

```
>>> Publisher.objects.all()
[<Publisher: Apress>, <Publisher: O'Reilly>]
```

这基本上相当于下述 SQL 语句：

```
SELECT id, name, address, city, state_province, country, website
FROM books_publisher;
```

> **提示**
>
> 注意，查找数据时，Django 使用的不是 `SELECT *`，而是把所有字段列出来。这样做是有原因的：某些情况下，`SELECT *` 较慢，而（更重要的是）列出字段更接近“Zen of Python”（Python 之禅）中的一个原则——“Explicit is better than implicit”（明了胜于晦涩）。如果想查看 Python 之禅的更多内容，在 Python 提示符中输入 `import this`。

下面详细分析 `Publisher.objects.all()` 这行代码：

- 首先，`Publisher` 是我们定义的模型。这没什么可意外的，想查找数据就应该使用相应的模型。
- 然后，访问 `objects` 属性。这叫*管理器*（manager），在第 9 章详述。现在，你只需知道，管理器负责所有“表层”数据操作，包括（最重要的）数据查询。所有模型都自动获得一个 `objects` 管理器，需要查询模型实例时都要使用它。
- 最后，调用 `all()` 方法。这是 `objects` 管理器的一个方法，返回数据库中的所有行。虽然返回的对象看似一个列表，但其实是一个查询集合（QuerySet）——表示数据库中一系列行的对象。附录 C 将详细说明查询集合。本章都将把它视作它所模仿的列表。

所有数据库查找操作都遵守这种一般模式，即在依附于模型的管理器上调用方法，执行相应的查询。

4.5.4 过滤数据

当然，我们很少需要一次性从数据库中选择所有数据。多数情况下，我们只想处理数据的子集。在 Django API 中，可以使用 `filter()` 方法过滤数据：

```
>>> Publisher.objects.filter(name='Apress')
[<Publisher: Apress>]
```

`filter()` 的关键字参数转换成相应的 SQL `WHERE` 子句。上述示例得到的 SQL 语句如下：

```
SELECT id, name, address, city, state_province, country, website
FROM books_publisher
WHERE name = 'Apress';
```

可以把多个参数传给 `filter()` 方法，进一步收窄要查询的数据：

```
>>> Publisher.objects.filter(country="U.S.A.",
state_province="CA")
[<Publisher: Apress>]
```

多个参数转换成 SQL `AND` 子句。因此，上述示例中的代码片段得到的 SQL 语句如下：

```
SELECT id, name, address, city, state_province, country, website
```

```
FROM books_publisher
WHERE country = 'U.S.A.'
AND state_province = 'CA';
```

注意，查找操作默认使用 SQL = 运算符做精确匹配查找。其他查找类型还有：

```
>>> Publisher.objects.filter(name__contains="press")
[<Publisher: Apress>]
```

注意，name 和 contains 之间有两个下划线。与 Python 一样，Django 使用双下划线表示“魔法”操作。这里，Django 把 __contains 部分转换成 SQL LIKE 语句：

```
SELECT id, name, address, city, state_province, country, website
FROM books_publisher
WHERE name LIKE '%press%';
```

支持的其他查找类型有：icontains（不区分大小写的 LIKE），startswith 和 endswith，以及 range（SQL BETWEEN 语句）。附录 B 详细说明这些查找类型。

4.5.5 检索单个对象

上述 filter() 示例都返回一个查询集合（可视作列表）。有时，较之列表，更适合获取单个对象。此时应该使用 get() 方法：

```
>>> Publisher.objects.get(name="Apress")
<Publisher: Apress>
```

这个方法只返回一个对象，而不是一个列表（更确切地说是查询集合）。因此，得到多个对象的查询会导致异常：

```
>>> Publisher.objects.get(country="U.S.A.")
Traceback (most recent call last):
    ...
MultipleObjectsReturned: get() returned more than one Publisher -- it returned 2! Loo\
kup parameters were {'country': 'U.S.A.'}
```

不返回对象的查询也导致异常：

```
>>> Publisher.objects.get(name="Penguin")
Traceback (most recent call last):
    ...
DoesNotExist: Publisher matching query does not exist.
```

DoesNotExist 异常是模型类的属性——Publisher.DoesNotExist。在应用程序中可以像下面这样捕获这个异常：

```
try:
    p = Publisher.objects.get(name='Apress')
except Publisher.DoesNotExist:
    print ("Apress isn't in the database yet.")
else:
    print ("Apress is in the database.")
```

4.5.6 排序数据

在运行上述示例的过程中你可能发现了，返回的对象好像是随机排序的。你想的没错，目前我们没有告诉数据库如何排序结果，因此数据使用数据库选择的顺序排序。在 Django 应用程序中，你可能想根据特定值排序结果，例如按字母表顺序。为此，使用 order_by() 方法：

```
>>> Publisher.objects.order_by("name")
[<Publisher: Apress>, <Publisher: O'Reilly>]
```

结果好像与之前的 all() 示例没有什么区别，不过现在 SQL 中指定了顺序：

```
SELECT id, name, address, city, state_province, country, website
FROM books_publisher
ORDER BY name;
```

可以根据任何字段排序：

```
>>> Publisher.objects.order_by("address")
 [<Publisher: O'Reilly>, <Publisher: Apress>]

>>> Publisher.objects.order_by("state_province")
 [<Publisher: Apress>, <Publisher: O'Reilly>]
```

如果想根据多个字段排序（以第一个字段排不出顺序时使用第二个字段），提供多个参数：

```
>>> Publisher.objects.order_by("state_province", "address")
 [<Publisher: Apress>, <Publisher: O'Reilly>]
```

此外，还可以反向排序。方法是在字段名称前面加上“-”（减号）：

```
>>> Publisher.objects.order_by("-name")
[<Publisher: O'Reilly>, <Publisher: Apress>]
```

虽然 order_by() 有一定的灵活性，但是每次都调用它相当繁琐。多数时候，我们始终使用同一个字段排序。此时，可以在模型中指定默认排序：

```
class Publisher(models.Model):
    name = models.CharField(max_length=30)
    address = models.CharField(max_length=50)
    city = models.CharField(max_length=60)
    state_province = models.CharField(max_length=30)
    country = models.CharField(max_length=50)
    website = models.URLField()

    def __str__(self):
        return self.name

    class Meta:
        ordering = ['name']
```

这里出现了一个新概念，内嵌在 Publisher 类定义体中的 class Meta（即要缩进，放在 class Publisher 内部）。任何模型都可以使用 Meta 类指定多个针对所在模型的选项。全部可用的选项参见附录 B，现在我们关注的是排序选项。指定这个选项后，使用 Django 数据库 API 检索 Publisher 对象时，如果没有明确调用 order_by()，都按照 name 字段排序。

4.5.7 链式查找

我们知道如何过滤数据，也知道如何排序数据了。当然，这两个操作经常需要一起做。此时，可以把查找“链接”在一起：

```
>>> Publisher.objects.filter(country="U.S.A.").order_by("-name")
[<Publisher: O'Reilly>, <Publisher: Apress>]
```

正如你所期待的那样，得到的 SQL 查询中既有 `WHERE` 子句，也有 `ORDER BY` 子句：

```
SELECT id, name, address, city, state_province, country, website
FROM books_publisher
WHERE country = 'U.S.A'
ORDER BY name DESC;
```

4.5.8 切片数据

另一个常见的需求是只查找固定数量的行。假如数据库中有几千个出版社记录，但是只想显示第一个。为此，可以使用 Python 标准的列表切片句法：

```
>>> Publisher.objects.order_by('name')[0]
<Publisher: Apress>
```

得到的 SQL 语句基本如下：

```
SELECT id, name, address, city, state_province, country, website
FROM books_publisher
ORDER BY name
LIMIT 1;
```

类似地，可以使用 Python 的范围切片句法检索数据子集：

```
>>> Publisher.objects.order_by('name')[0:2]
```

这样得到的是两个对象，基本上相当于下述 SQL 语句：

```
SELECT id, name, address, city, state_province, country, website
FROM books_publisher
ORDER BY name
OFFSET 0 LIMIT 2;
```

注意，不支持使用负数：

```
>>> Publisher.objects.order_by('name')[-1]
Traceback (most recent call last):
  ...
AssertionError: Negative indexing is not supported.
```

不过，这容易解决。只需修改 `order_by()` 语句，如下所示：

```
>>> Publisher.objects.order_by('-name')[0]
```

4.5.9 在一个语句中更新多个对象

4.5.2 节说过，`save()` 方法更新一行中的所有列。然而，你的应用程序可能只需要更新部分列。比如说，你可

能想把 Apress 出版社的名称由 'Apress' 改为 'Apress Publishing'。使用 save() 的代码如下：

```
>>> p = Publisher.objects.get(name='Apress')
>>> p.name = 'Apress Publishing'
>>> p.save()
```

这基本上相当于下述 SQL 语句：

```
SELECT id, name, address, city, state_province, country, website
FROM books_publisher
WHERE name = 'Apress';

UPDATE books_publisher SET
    name = 'Apress Publishing',
    address = '2855 Telegraph Ave.',
    city = 'Berkeley',
    state_province = 'CA',
    country = 'U.S.A.',
    website = 'http://www.apress.com'
WHERE id = 52;
```

（注意，这个示例假定 Apress 的 ID 为 52。）从这个示例可以看出，Django 的 save() 方法会设定所有列的值，而不是只设定 name 列的值。如果你所处的环境可能同时由其他操作修改其他列，最好只更新需要修改的值。为此，使用 QuerySet 对象的 update() 方法。例如：

```
>>> Publisher.objects.filter(id=52).update(name='Apress Publishing')
```

这样得到的 SQL 语句更高效，而且不会导致条件竞争：

```
UPDATE books_publisher
SET name = 'Apress Publishing'
WHERE id = 52;
```

update() 方法可以在任何 QuerySet 对象上调用，这意味着可以通过它批量编辑多个记录。下述代码把每个 Publisher 记录的 country 列都由 'U.S.A.' 改为 'USA'：

```
>>> Publisher.objects.all().update(country='USA')
2
```

update() 方法有返回值，是一个整数，表示修改的记录数量。在上述示例中，返回值是 2。

4.5.10 删除对象

若想从数据库中删除一个对象，只需在对象上调用 delete() 方法：

```
>>> p = Publisher.objects.get(name="O'Reilly")
>>> p.delete()
>>> Publisher.objects.all()
[<Publisher: Apress Publishing>]
```

此外，还可以在任何 QuerySet 对象上调用 delete() 方法，批量删除对象。这与前一节所讲的 update() 方法类似。

```
>>> Publisher.objects.filter(country='USA').delete()
>>> Publisher.objects.all().delete()
```

```
>>> Publisher.objects.all()
[]
```

删除数据要谨慎！为了防止不小心把表中的数据都删除，想删除表中的一切数据时，Django 要求必须显式调用 all() 方法。例如，下述代码无效：

```
>>> Publisher.objects.delete()
Traceback (most recent call last):
  File "", line 1, in
AttributeError: 'Manager' object has no attribute 'delete'
```

添加 all() 方法后就可以了：

```
>>> Publisher.objects.all().delete()
```

如果只想删除部分数据，无需调用 all() 方法。下面再以前面的一个示例为例：

```
>>> Publisher.objects.filter(country='USA').delete()
```

4.6 接下来

读完本章后，你掌握了足够的 Django 模型知识，可以编写基本的数据库应用程序了。第 9 章将说明 Django 数据库层的一些高级用法。定义好模型之后，接下来应该使用数据填充数据库。你可能有旧数据，此时可以阅读第 21 章，了解如何集成旧数据库。你可能依靠网站的用户提供数据，此时可以阅读第 6 章，学习如何处理用户提交的表单数据。但是，有些情况下，你或者你所在的团队要自己动手录入数据，此时最好有一个 Web 界面，用于输入和管理数据。下一章介绍的 Django 管理界面正是为这种情况而生的。

第 5 章 Django 管理后台

对多数现代的网站而言，管理界面是基础设施的重要组成部分。这是一个 Web 界面，限制只让授信的网站管理员访问，用于添加、编辑和删除网站内容。常见的示例有：发布博客的界面，网站管理人员审核用户评论的后端，客户用来更新新闻稿的工具。

不过，管理界面有个问题：构建起来繁琐。构建面向公众的功能时，Web 开发是有趣的，但是管理界面一成不变，要验证用户身份、显示并处理表单、验证输入，等等。这个过程无趣、乏味。

那么，对这样一个无趣、乏味的任务，Django 采取了什么措施呢？它为你代劳了。

在 Django 中，构建管理界面不算个事。本章探讨 Django 自动生成的管理界面，了解它为模型提供的便利界面，以及可以使用的其他功能。

5.1 使用 Django 管理后台

在第 1 章中运行 `django-admin startproject mysite` 时，Django 为我们创建并配置了默认的管理后台。我们只需创建一个管理员用户（超级用户），就可以登录管理后台。

> **提示**
>
> 如果你使用 Visual Studio，无需在命令行中完成接下来的步骤，在 Visual Studio 的"项目"菜单中添加一个超级用户即可。

执行下述命令，创建一个管理员用户：

```
python manage.py createsuperuser
```

输入想用的用户名，然后按回车键：

```
Username: admin
```

接下来会提示你输入电子邮件地址：

```
Email address: admin@example.com
```

最后，输入密码。密码要输入两次，第二次是对第一次的确认。

```
Password: **********
Password (again): *********
Superuser created successfully.
```

5.1.1 启动开发服务器

在 Django 1.8 中，管理后台默认已激活。下面启动开发服务器，用一下它。读过前面几章我们知道，启动开发服务器的方法如下：

```
python manage.py runserver
```

现在，打开 Web 浏览器，访问本地域名上的 /admin/ 路径，例如 http://127.0.0.1:8000/admin/。你应该看到管理后台的登录界面（图 5-1）。

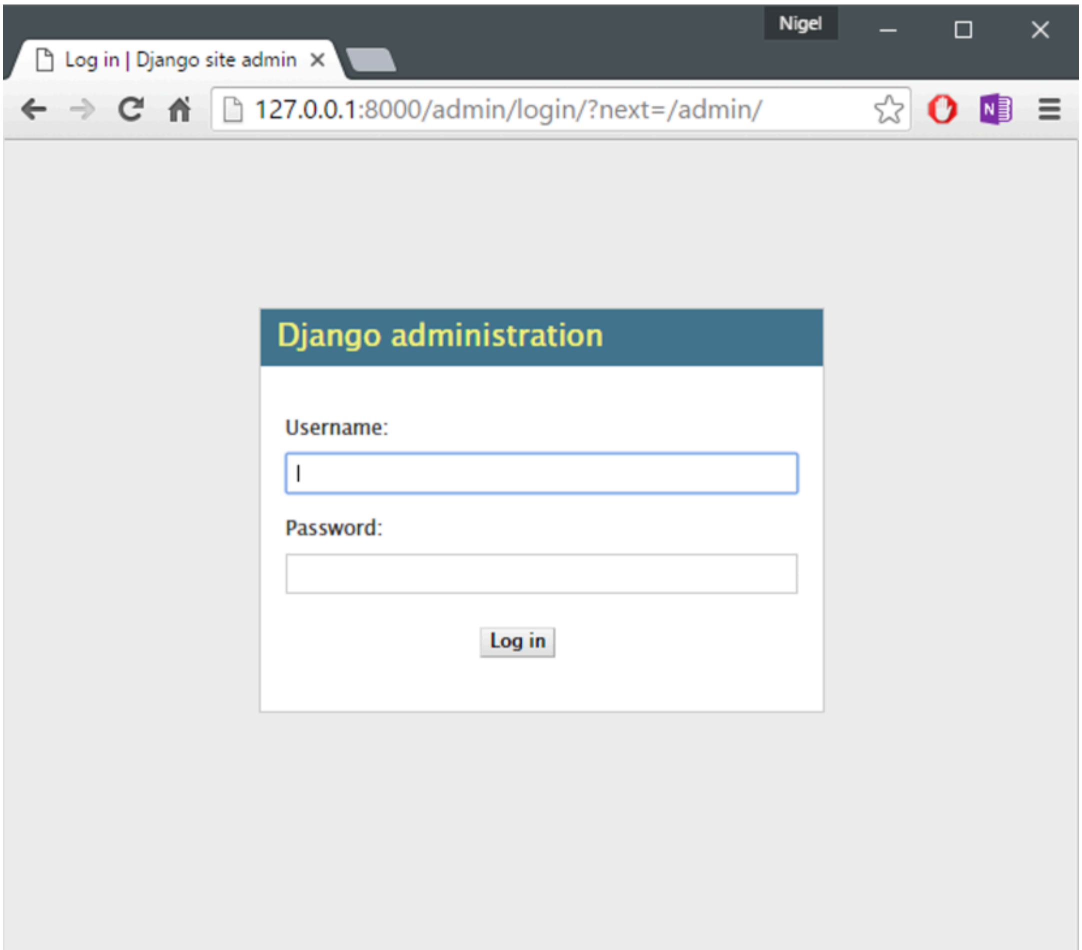


图 5-1：Django 管理后台的登录界面

因为默认启用了本地化，所以登录界面可能会使用你所选的语言显示，当然，这取决于浏览器的设置和 Django 是否有那门语言的翻译。

5.1.2 进入管理后台

现在，使用前面创建的超级用户账户登录。登录后应该看到 Django 管理后台的首页（图 5-2）。

你应该看到两种可编辑的内容：Groups（分组）和 Users（用户）。这是 Django 自带的身份验证框架 django.contrib.auth 提供的。管理后台主要供非技术人员使用，因此无需过多解释。尽管如此，我们还是要快速介绍一些基本功能。

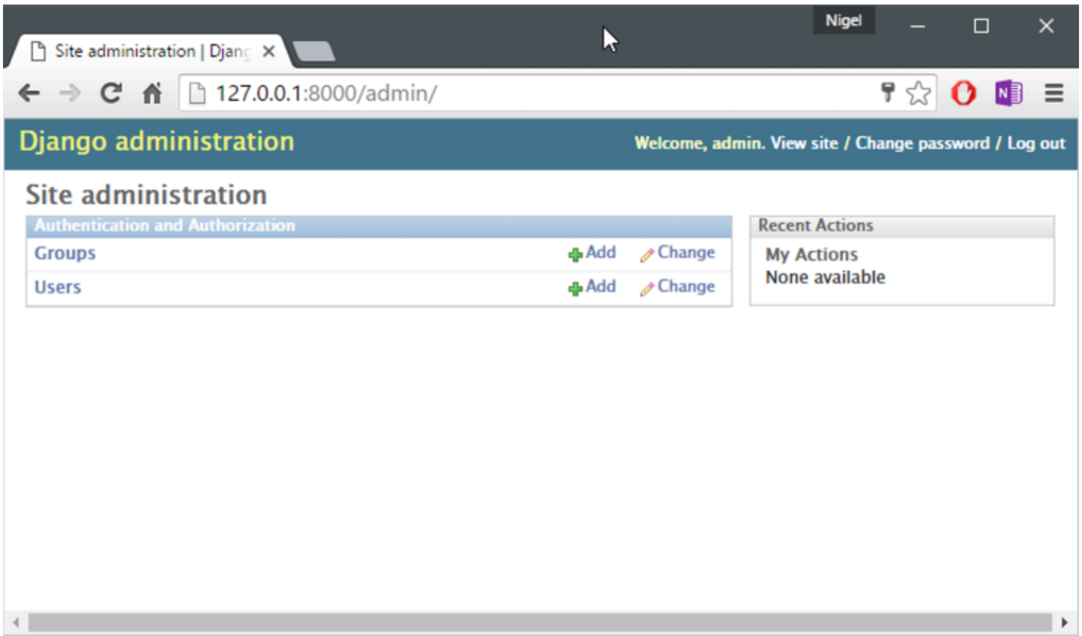


图 5-2：Django 管理后台的首页

Django 中每种数据都有一个修改列表和编辑表单。前者列出数据库中所有可用的对象，后者用于添加、修改或删除数据库中的特定记录。点击“Users”那一行里的“Change”链接，加载用户的修改列表（图 5-3）。

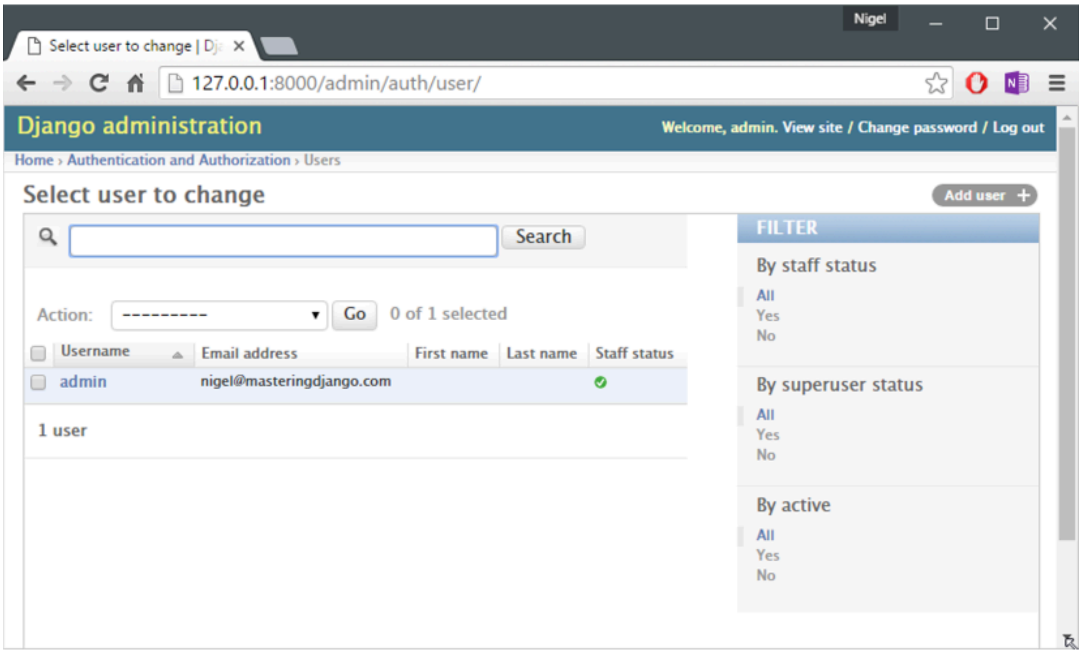


图 5-3：用户的修改列表页面

这个页面显示数据库中的所有用户，你可以把它想象成 SQL 查询 `SELECT * FROM auth_user;` 的精美 Web 版本。如果你一直跟着书中的示例做，会看到一个用户；添加更多的用户后，你会发现过滤、排序和搜索功能很有用。

过滤功能在右边，排序通过点击表头实现；搜索框在顶部，可以通过用户名搜索。点击用户的用户名后会看到编辑用户的表单（图 5-4）。

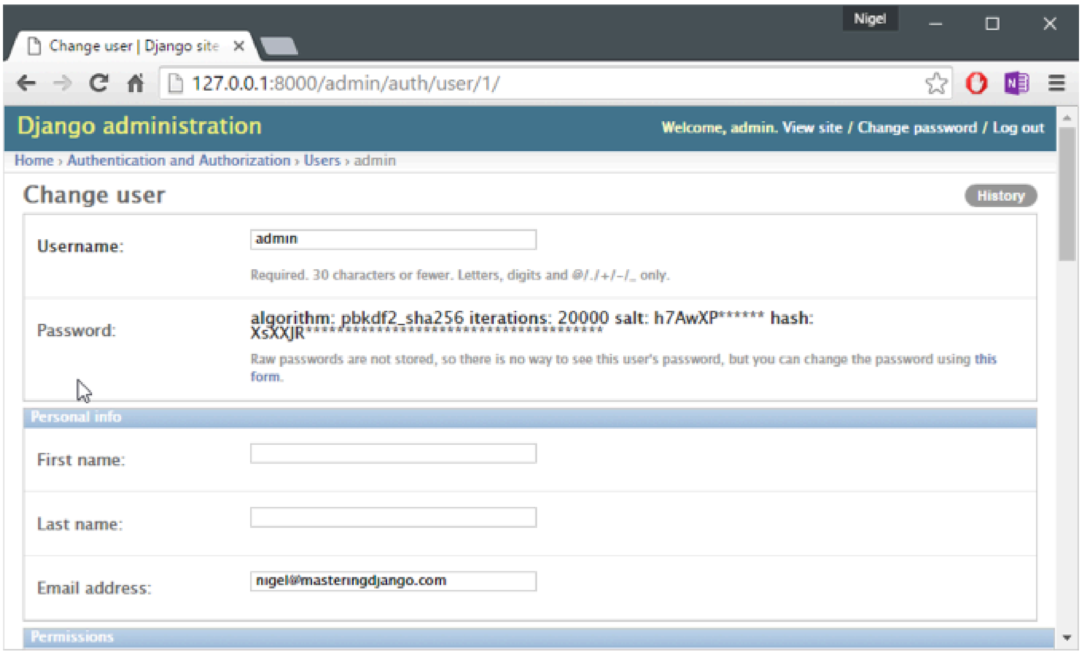


图 5-4：编辑用户的表单

在这个页面中可以修改用户的属性，例如姓名和各种权限。注意，如果想修改用户的密码，点击密码字段下面的链接，不能直接修改哈希密码。

还要注意，不同字段类型显示的小组件不同，例如，日期（时间）字段显示的是日历控件、布尔值字段显示的是复选框、字符字段显示的是文本输入框。

如果想删除记录，点击编辑表单左下角的“Delete”按钮。点击那个按钮后会显示确认页面，有时那个页面还会显示要删除的依赖对象。（比如说删除出版社记录时，那个出版社名下的图书也将被删除！）

如果想添加记录，在管理后台首页点击相应行里的“Add”链接。此时显示一个空表单，让你填写数据。

注意，管理界面还能验证输入。留空必填的字段，或者在日期字段中输入无效的日期试试，保存时会看到类似图 5-5 中的错误。

编辑现有对象时，窗口右上角有个“History”链接。通过管理界面所做的每个改动都记录在案，点击“History”链接便可查看（图 5-6）。

旁注 5-1：管理后台的运作方式

管理后台在背后是如何工作的呢？相当简单。启动服务器时，Django 运行 `admin.autodiscover()` 函数。在早期的 Django 版本中，要在 `urls.py` 文件中调用这个函数，但是现实 Django 会自动运行它。这个函数迭代 `INSTALLED_APPS` 设置，在安装的各个应用中查找一个名为 `admin.py` 的文件。如果应用中存在这个文件，就执行里面的代码。在 books 应用的 `admin.py` 文件中，我们调用 `admin.site.register()`，在管理后台中注册各个模型。只有注册的模型才能在管理后台中显示。`django.contrib.auth` 应用也有 `admin.py` 文件，因此管理界面中才显示有“Users”和“Groups”。其他 `django.contrib` 应用，如 `django.contrib.redirects`，也把自己添加到管理后台中，从网上下载的很多第三方 Django 应用程序也会这么做。其实，Django 管理后台也是一个 Django 应用程序，有自己的模型、模板、视图和 URL 模式。你的应用之所以有管理后台，是因为你在 URL 配置中设置了——这与设置自己编写的视图一样。你可以在你下

载的 Django 代码基中查看 `django/contrib/admin` 里的代码，查看它的模板、视图和 URL 配置。但是，不要直接修改任何代码，因为里面有众多定制管理后台的钩子。阅读 Django 管理后台的代码时记住一点，在读取模型的元数据方面它做了相当复杂的操作，所以可能要花点时间才能理解。

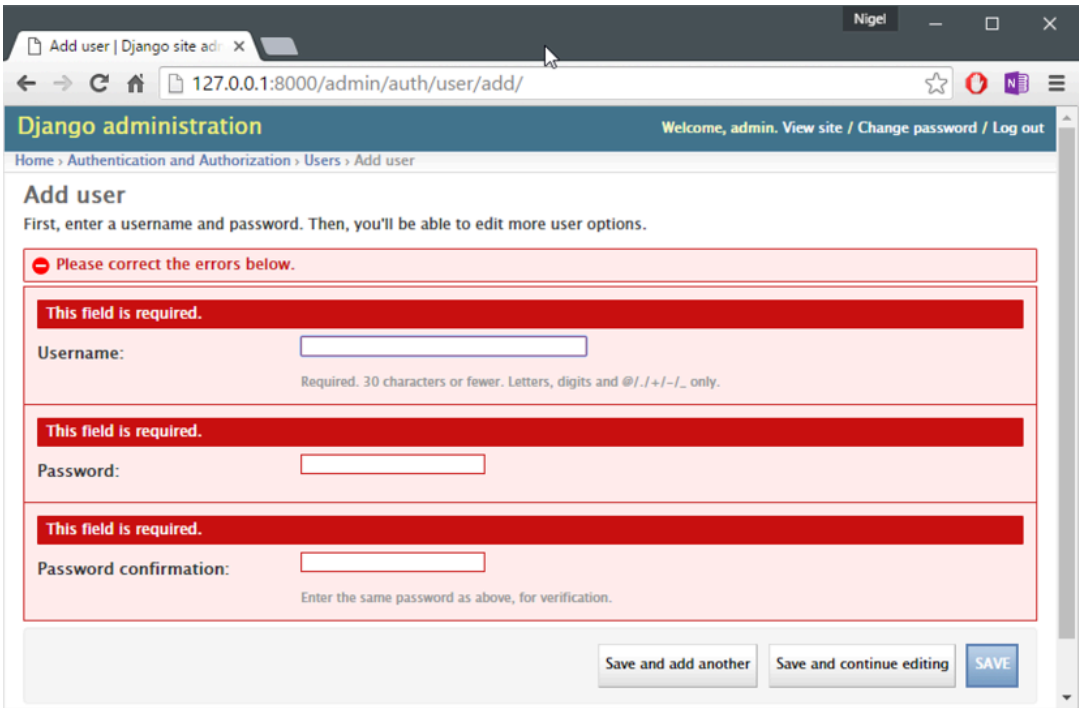


图 5-5：显示有错误的编辑表单

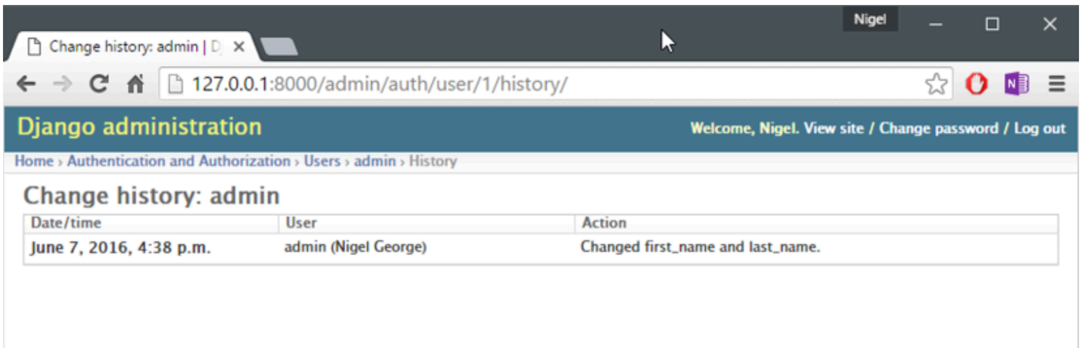


图 5-6：一个对象的修改历史页面

5.2 把模型添加到 Django 管理后台中

有一个重要操作我们还没做。我们要把自己编写的模型添加到管理后台中，这样便可以在精美的界面中添加、修改和删除自定义数据表中的对象。我们继续以第 4 章开发的 `books` 应用为例。那个应用定义了三个模型：`Publisher`、`Author` 和 `Book`。`startapp` 命令应该在 `books` 目录（`mysite/books`）中创建了 `admin.py` 文件，如果没有，自己动手创建，然后输入下面几行代码：

```
from django.contrib import admin
from .models import Publisher, Author, Book
```

```
admin.site.register(Publisher)
admin.site.register(Author)
admin.site.register(Book)
```

上述代码告诉 Django 管理后台，为这几个模型提供界面。添加代码之后，在 Web 浏览器中访问管理后台首页（`http://127.0.0.1:8000/admin/`），你应该会看到一个“Books”表，列出 Authors、Books 和 Publishers 链接。（可能要重启开发服务器改动才能生效。）现在，这三个模型的管理界面完全可用了。很简单吧！

花点时间添加和修改记录，向数据库中填充一些数据。如果你跟着第 4 章的示例做，创建了几个 `Publisher` 对象（而且没删除），在修改列表页面会看到那些出版社记录。

有个功能值得提一下：管理后台能自动处理外键和多对多关系（`Book` 模型中都有）。下面回顾一下 `Book` 模型的代码：

```
class Book(models.Model):
    title = models.CharField(max_length=100)
    authors = models.ManyToManyField(Author)
    publisher = models.ForeignKey(Publisher)
    publication_date = models.DateField()

    def __str__(self):
        return self.title
```

在 Django 管理后台的添加图书页面（`http://127.0.0.1:8000/admin/books/book/add/`），出版社字段（`ForeignKey`）显示为选择框，作者字段（`ManyToManyField`）显示为多选框。这两个字段旁边都有一个绿色加号图标，用于添加相应的记录。

比如说，点击“Publisher”字段旁边的绿色加号按钮后会弹出一个窗口，用于添加出版社记录。在弹出窗口中成功创建出版社记录后，添加图书表单会更新，列出新创建的出版社。真是太棒了！

5.3 把字段设为可选的

在管理后台中操作一会之后，你可能会发现有个局限：编辑表单要求填写每个字段，而有时候某些字段需要是可选的。比如说，我们可能想让 `Author` 模型的 `email` 字段可选，即允许使用空字符串。事实也是如此，你不可能有每位作者的电子邮件地址。

为了把 `email` 字段设为可选的，我们要编辑 `Author` 模型（在 `mysite/books/models.py` 文件中），为 `email` 字段添加 `blank=True` 参数，如下所示：

```
class Author(models.Model):
    first_name = models.CharField(max_length=30)
    last_name = models.CharField(max_length=40)
    email = models.EmailField(blank=True)
```

这个参数告诉 Django，作者的电子邮件地址允许为空值。默认情况下，所有字段都设定了 `blank=False`，意即不允许为空值。

这里发生了一件有趣的事。截至目前，除了 `__str__()` 方法之外，模型的作用是定义数据库表，即与 SQL `CREATE TABLE` 语句等效的 Python 代码。添加 `blank=True` 之后，模型不再只用于定义数据库表的结构了。

现在，模型类的作用变得丰富了，不仅知道 `Author` 对象是什么，还知道它们能做什么。`email` 字段不仅表示数据库中的一个 `VARCHAR` 列，在 Django 管理后台等上下文中，它还是一个可选字段。

添加 blank=True 之后，再次访问“Add author”表单（http://127.0.0.1:8000/admin/books/author/add/），你会发现字段的标注——“Email”——不是粗体了。这表明，它不是必填字段了。现在添加作者，无需提供电子邮件地址；提交空值时，不会显示亮红色的“This field is required”消息。

5.3.1 把日期和数值字段设为可选的

为日期和数值字段设定 blank=True 时经常遇到问题，这背后涉及很多知识。SQL 使用是一个特殊的值表示空值——NULL。它的意思是“未知”或“无效”，或者其他情境中的特定意思。在 SQL 中，NULL 与空字符串不是一回事，就像 Python 对象 None 不是空字符串（""）一样。

因此，特定的字符字段（如 VARCHAR 列）的值既可以是 NULL，也可以是空字符串。这可能导致意料之外的歧义：“为什么在这个记录中是 NULL，而在其他记录中是空字符串？二者有区别吗，还是说输入的数据不一致？”以及：“如何获取所有为空值的记录，应该包含值为 NULL 和空字符的记录，还是只选择值为空字符串的记录？”

为了避免这种歧义，Django 自动生成的 CREATE TABLE 语句（参见第 4 章）中每个列定义都有 NOT NULL。例如，下面是为 Author 模型生成的语句，摘自第 4 章：

```
CREATE TABLE "books_author" (
    "id" serial NOT NULL PRIMARY KEY,
    "first_name" varchar(30) NOT NULL,
    "last_name" varchar(40) NOT NULL,
    "email" varchar(75) NOT NULL
);
```

多数情况下，这种默认行为对应用程序来说是最佳选择，无需费力处理数据不一致。而且，Django 的其他部分能很好地支持这个行为，例如 Django 管理后台，当你留空字符字段时，Django 插入数据库的是空字符串（而不是 NULL 值）。

但是，对空字符串不是有效值的数据库列类型（如日期、时间和数字）来说，这样处理不行。如果把空字符串插入日期或整数列，数据库有可能报错——这取决于你用的数据库。（PostgreSQL 严格，遇到这种情况时抛出异常；MySQL 可能允许这么做，也可能不允许，根据版本、时间和月相而定。）

此时，只能使用 NULL 指定空值。在 Django 模型中，指定接受 NULL 值的方式是为字段设定 null=True 参数。因此，说起来有点复杂：如果想让日期字段（如 DateField、TimeField、DateTimeField）或数值字段（如 IntegerField、DecimalField、FloatField）接受空值，要同时添加 null=True 和 blank=True。

下面通过实例说明。我们来修改 Book 模型，允许 publication_date 字段为空。修改后的代码如下：

```
class Book(models.Model):
    title = models.CharField(max_length=100)
    authors = models.ManyToManyField(Author)
    publisher = models.ForeignKey(Publisher)
    publication_date = models.DateField(blank=True, null=True)
```

添加 null=True 比添加 blank=True 复杂，因为前者修改了数据库的语义，即修改了 CREATE TABLE 语句，把 publication_date 字段的 NOT NULL 删掉了。为了完成修改，我们要更新数据库。基于一些原因，Django 不会试图自动修改数据库模式，所以每次对模型做这种修改之后要自己动手执行 python manage.py migrate 命令。执行完迁移之后，回到管理后台，现在添加图书表单应该允许把出版日期设为空值了。

5.4 自定义字段的标注

在管理后台的编辑表单中，各个字段的标注根据模型中字段的名称生成。生成方式很简单：把下划线替换成空格，再把第一个字母变成大写。例如，Book 模型中 publication_date 字段对应的标注是“Publication date”。

然而，根据字段名称并不是总能生成好的标注，因此有时需要自定义。自定义标注的方式是为模型字段指定 verbose_name 参数。例如，下述代码把 Author.email 字段的标注改为“e-mail”（中间有个连字符）：

```
class Author(models.Model):
    first_name = models.CharField(max_length=30)
    last_name = models.CharField(max_length=40)
    email = models.EmailField(blank=True, verbose_name='e-mail')
```

修改之后，再次访问编辑作者表单，你会发现显示的是新标注。注意，除非始终应该为大写（如 "USA state"），否则不应该把 verbose_name 值的首字母设为大写。如果需要，Django 会自动把首字母变成大写，在不需要大写的地方，则直接使用 verbose_name 的值。

5.5 自定义 ModelAdmin 类

目前我们所做的改动，添加 blank=True、null=True 和 verbose_name，修改的其实都是模型层，只是碰巧管理后台有用到，还未涉及管理后台自身。

除此之外，Django 管理后台也提供了丰富的选项，可以定制处理具体模型的方式。这些选项在 ModelAdmin 类中，这些类包含特定管理后台实例中特定模型的配置。

5.5.1 自定义修改列表

我们将指定 Author 模型的修改列表中显示的字段，以此说明如何定制管理后台。默认情况下，修改列表显示的是各个对象的 __str__() 方法返回的结果。在第 4 章，我们为 Author 对象定义了 __str__() 方法，显示名字和姓：

```
class Author(models.Model):
    first_name = models.CharField(max_length=30)
    last_name = models.CharField(max_length=40)
    email = models.EmailField(blank=True, verbose_name ='e-mail')

    def __str__(self):
        return u'%s %s' % (self.first_name, self.last_name)
```

因此，Author 对象的修改列表中显示各个作者的名字和姓，如图 5-7 所示。

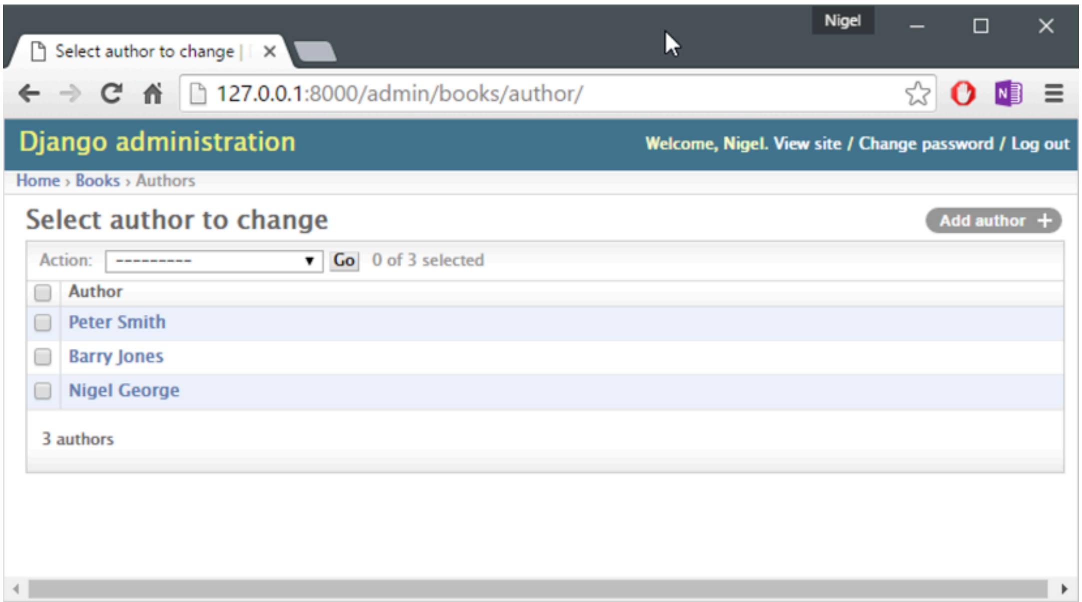


图 5-7：作者的修改列表页面

我们可以改进这种默认行为，在修改列表中添加几个其他字段。比如说，可以在列表中显示作者的电子邮件地址；另外，如果能按照名字和姓排序就好了。为此，要为 Author 模型定义一个 ModelAdmin 子类。这个类是定制管理后台的关键，其中最基本的一件事是指定修改列表页面显示的字段。参照下述代码修改 admin.py 文件：

```
from django.contrib import admin
from mysite.books.models import Publisher, Author, Book

class AuthorAdmin(admin.ModelAdmin):
    list_display = ('first_name', 'last_name', 'email')

admin.site.register(Publisher)
admin.site.register(Author, AuthorAdmin)
admin.site.register(Book)
```

我们做了以下几件事：

- 定义 AuthorAdmin 类。它是 django.contrib.admin.ModelAdmin 的子类，存放指定模型在管理后台中的自定义配置。我们只做了一项定制，list_display，把它的值设为一个元组，指定要在修改列表页面显示的字段名称。当然，模型中必须有这些字段。
- 修改 admin.site.register() 调用，在 Author 后面添加 AuthorAdmin。你可以把这行代码理解为“以 AuthorAdmin 中的选项注册 Author 模型”。admin.site.register() 函数的第二个参数可选，其值是一个 ModelAdmin 子类。如果不指定第二个参数（Publisher 和 Book 模型就是这样），Django 使用默认选项注册模型。

修改之后，刷新作者的修改列表页面，你会看到现在显示了三列：名字、姓和电子邮件地址。此外，点击这三列的表头都可以排序各列（见图 5-8）。

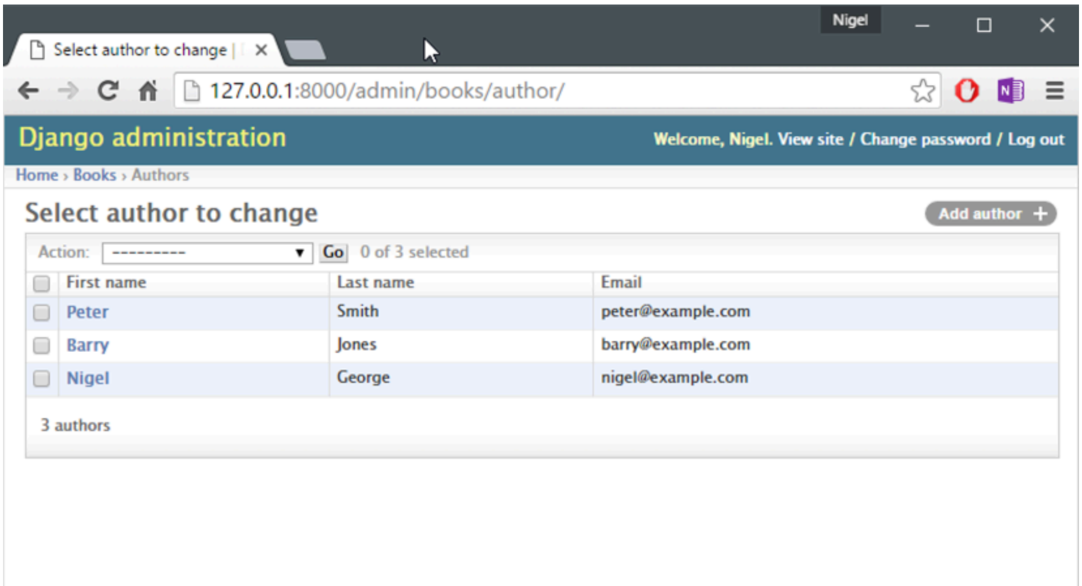


图 5-8：添加 list_display 之后的作者修改列表页面

接下来，添加一个简单的搜索框。在 AuthorAdmin 类中添加 search_fields：

```
class AuthorAdmin(admin.ModelAdmin):
    list_display = ('first_name', 'last_name', 'email')
    search_fields = ('first_name', 'last_name')
```

刷新浏览器中的页面，应该会在页面顶部看到一个搜索框（图 5-9）。我们刚刚添加的代码让管理后台的修改列表页面增加一个搜索框，用于搜索 first_name 和 last_name 字段。正如用户期待的那样，这个搜索框不区分大小写，而且两个字段都搜索。假如搜索字符串“bar”，会搜到名字为 Barney 的作者和姓为 Hobarson 的作者。

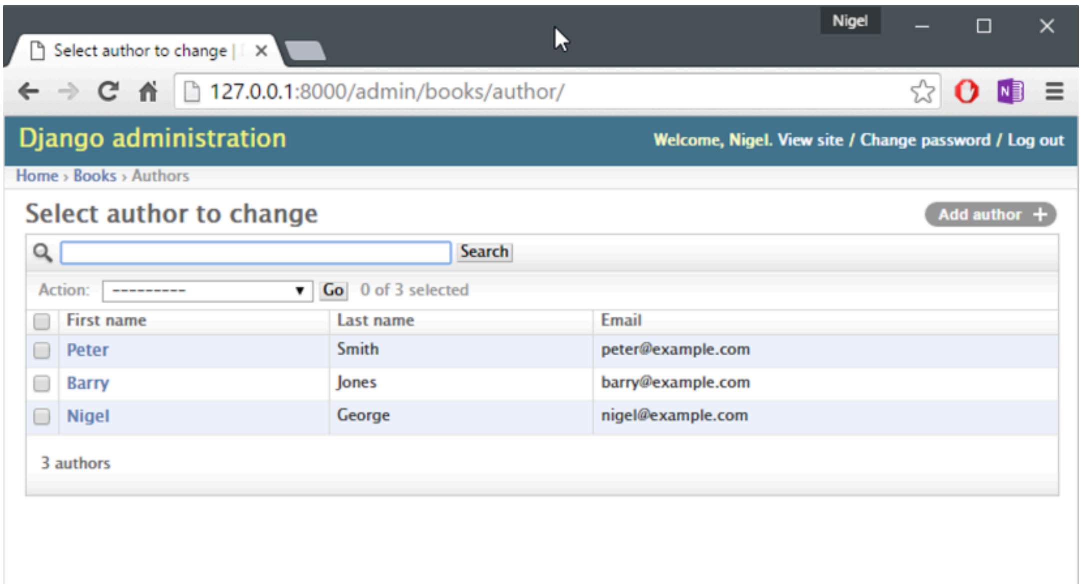


图 5-9：添加 search_fields 之后的作者修改列表页面

下面为 Book 模型的修改列表页面添加几个日期过滤器：

```python
from django.contrib import admin
from mysite.books.models import Publisher, Author, Book

class AuthorAdmin(admin.ModelAdmin):
    list_display = ('first_name', 'last_name', 'email')
    search_fields = ('first_name', 'last_name')

class BookAdmin(admin.ModelAdmin):
    list_display = ('title', 'publisher', 'publication_date')
    list_filter = ('publication_date',)

admin.site.register(Publisher)
admin.site.register(Author, AuthorAdmin)
admin.site.register(Book, BookAdmin)
```

这一次定制的是另一个模型的选项，因此定义一个单独的 `ModelAdmin` 子类，`BookAdmin`。首先，定义 `list_display` 属性，让修改列表好看一些。然后，为 `list_filter` 属性赋值一个字段元组，在修改列表页面的右边创建过滤器。Django 为日期字段提供了几个便利的过滤器："Today"（今天）、"Past 7 days"（过去 7 天）、"This month"（本月）和"This year"（本年）——这些是 Django 的开发者觉得过滤日期时最常用的。这些过滤器如图 5-10 所示。

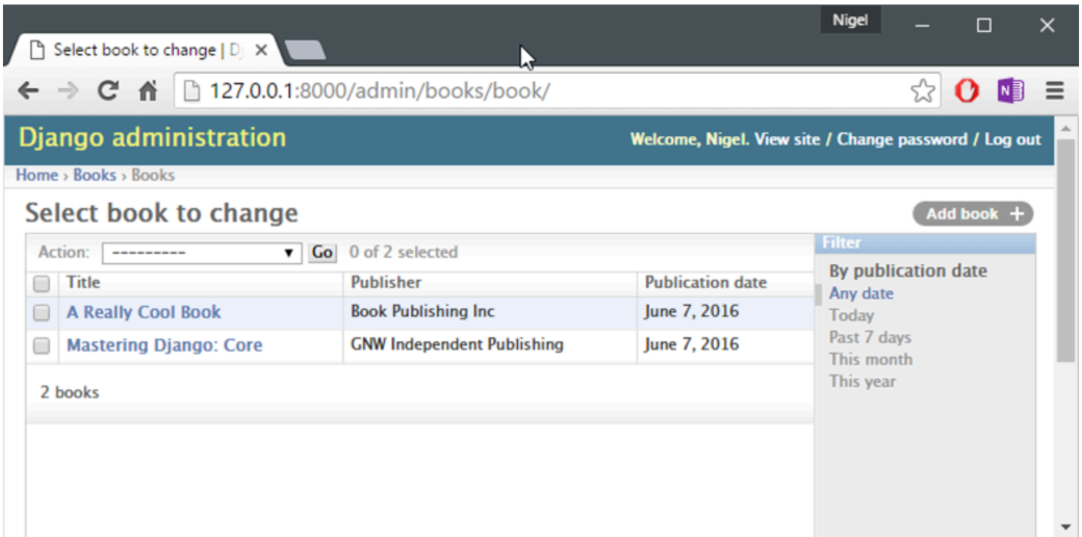


图 5-10：添加 `list_filter` 之后的图书修改列表页面

`list_filter` 也能处理其他类型的字段，而非 `DateField` 一个。（比如说，可以试试 `BooleanField` 和 `ForeignKey` 字段。）只要有超过两个值供选择，过滤器就会显示。提供日期过滤器的另一种方法是使用 `date_hierarchy` 选项，如下所示：

```python
class BookAdmin(admin.ModelAdmin):
    list_display = ('title', 'publisher','publication_date')
    list_filter = ('publication_date',)
    date_hierarchy = 'publication_date'
```

添加这个选项之后，修改列表上边会显示一个日期层级导航栏，如图 5-11 所示。这个导航栏先显示年份，然后向下显示月份和具体某一天。

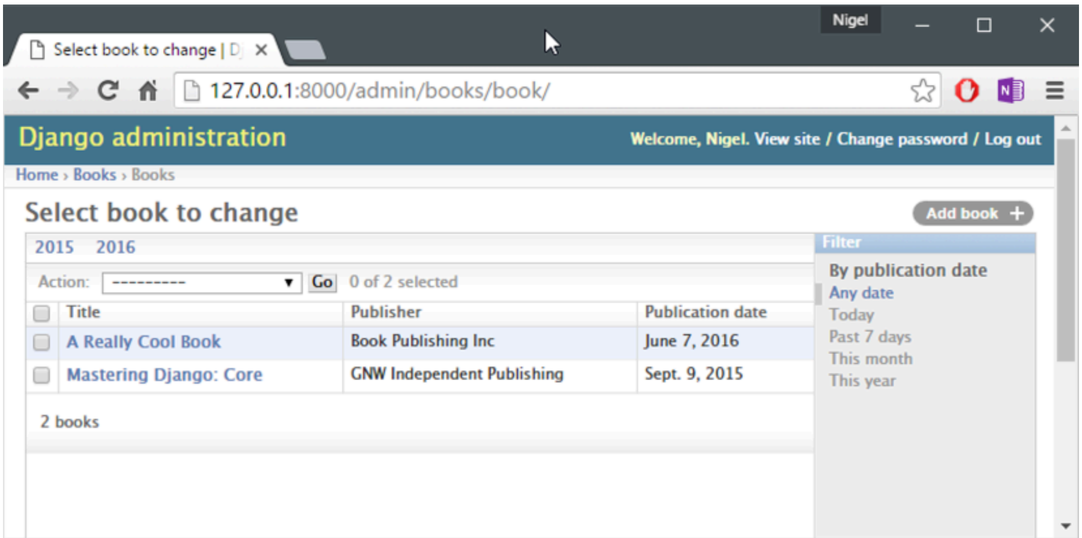


图 5-11：添加 date_hierarchy 后的图书修改列表页面

注意，date_hierarchy 的值是一个字符串，不是元组，因为只能使用一个日期字段创建层级导航。最后，修改默认的排序方式，让修改列表页面的图书始终以出版日期倒序排列。默认情况下，修改列表根据模型中 class Meta（参见第 4 章）里的 ordering 属性排序，不过我们没设定，因此没有顺序。

```
class BookAdmin(admin.ModelAdmin):
    list_display = ('title', 'publisher','publication_date')
    list_filter = ('publication_date',)
    date_hierarchy = 'publication_date'
    ordering = ('-publication_date',)
```

ModelAdmin 子类中的 ordering 选项与模型的 class Meta 中的 ordering 属性的作用完全一样，不过只使用列表中的第一个字段名称。ordering 选项的值是一个字段名称列表或元组，如果想倒序，加上减号即可。刷新图书修改列表，看看变化。注意，“Publication date”表头中有个小箭头，指明记录的排列顺序（图 5-12）。

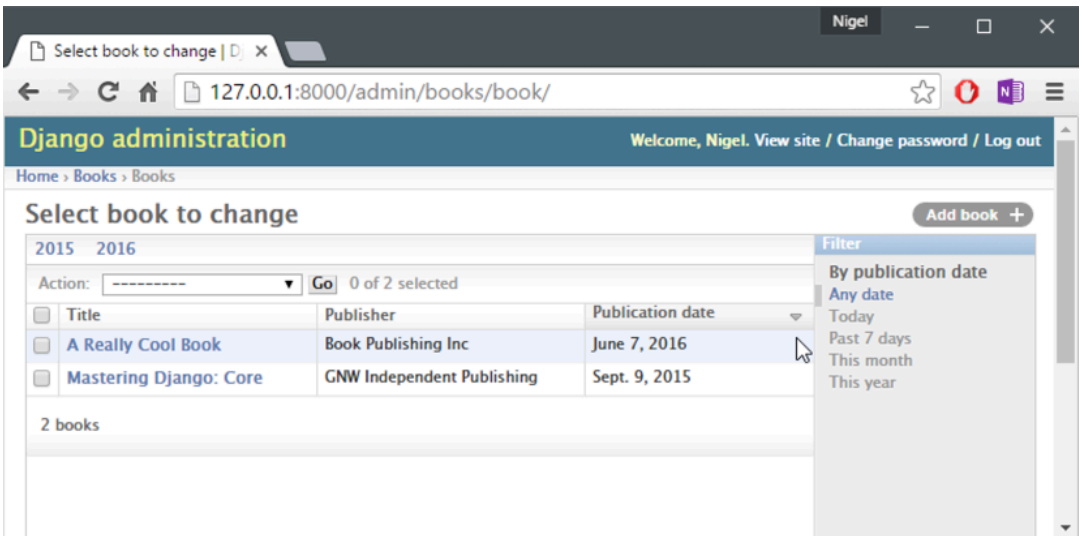


图 5-12：添加 ordering 后的图书修改列表页面

至此，我们介绍了主要的修改列表选项。通过这些选项，只需几行代码就能得到可在生产环境使用的强大的

数据编辑界面。

5.5.2 自定义编辑表单

与修改列表一样，编辑表单的很多方面也可以定制。首先，我们来定制字段的排序方式。默认情况下，编辑表单中的字段顺序与在模型中的定义顺序一致。我们可以在 ModelAdmin 子类中使用 fields 选项修改排序：

```
class BookAdmin(admin.ModelAdmin):
    list_display = ('title', 'publisher', 'publication_date')
    list_filter = ('publication_date',)
    date_hierarchy = 'publication_date'
    ordering = ('-publication_date',)
    fields = ('title', 'authors', 'publisher', 'publication_date')
```

这样修改之后，图书的编辑表单会使用指定的顺序显示各个字段。在书名后面显示作者稍微符合直觉。当然，字段的顺序应该根据输入数据的流程而定。世界上没有两个表单是相同的。

fields 选项还有一个作用：排除特定的字段，禁止编辑。只需去掉想排除的字段即可。如果你只相信管理员有能力编辑数据的某些部分，或者某些字段由外部的自动化流程修改，就可以这么做。

例如，在这个书籍数据库中，我们可以隐藏 publication_date 字段，禁止编辑：

```
class BookAdmin(admin.ModelAdmin):
    list_display = ('title', 'publisher','publication_date')
    list_filter = ('publication_date',)
    date_hierarchy = 'publication_date'
    ordering = ('-publication_date',)
    fields = ('title', 'authors', 'publisher')
```

这样修改之后，通过图书的编辑表单无法指定出版日期。如果你是编辑，不想让作者把出版日期往后退，就可以这么做。（当然，这纯粹是个虚构的例子。）用户使用这个不完整的表单添加新图书时，Django 会把 publication_date 设为 None，所以那个字段要指定 null=True 参数。

编辑表单中另一个经常需要定制的字段是多对多关系字段。前文已经说过，管理后台为多对多关系字段显示的是多项选择框，这是最符合逻辑的 HTML 输入控件，但是多项选择框不易使用。如果想选择多个项目，要按住 Ctrl 键（Mac 上的 Command 键）。

管理后台提供了解说文本，但是有几百个选项时还是不灵便。管理后台为此提供的解决方案是 filter_horizontal 选项。我们把它添加到 BookAdmin 中，看看效果：

```
class BookAdmin(admin.ModelAdmin):
    list_display = ('title', 'publisher','publication_date')
    list_filter = ('publication_date',)
    date_hierarchy = 'publication_date'
    ordering = ('-publication_date',)
    filter_horizontal = ('authors',)
```

（如果你一直跟着我做，注意，我们删除了 fields 选项，让所有字段都在编辑表单中显示出来。）刷新图书的编辑表单，你会看到“Authors”部分现在使用一个精美的 JavaScript 过滤器界面，可以动态搜索选项，把指定的作者从“Available authors”移到“Chosen authors”框中（以及反向移动）。

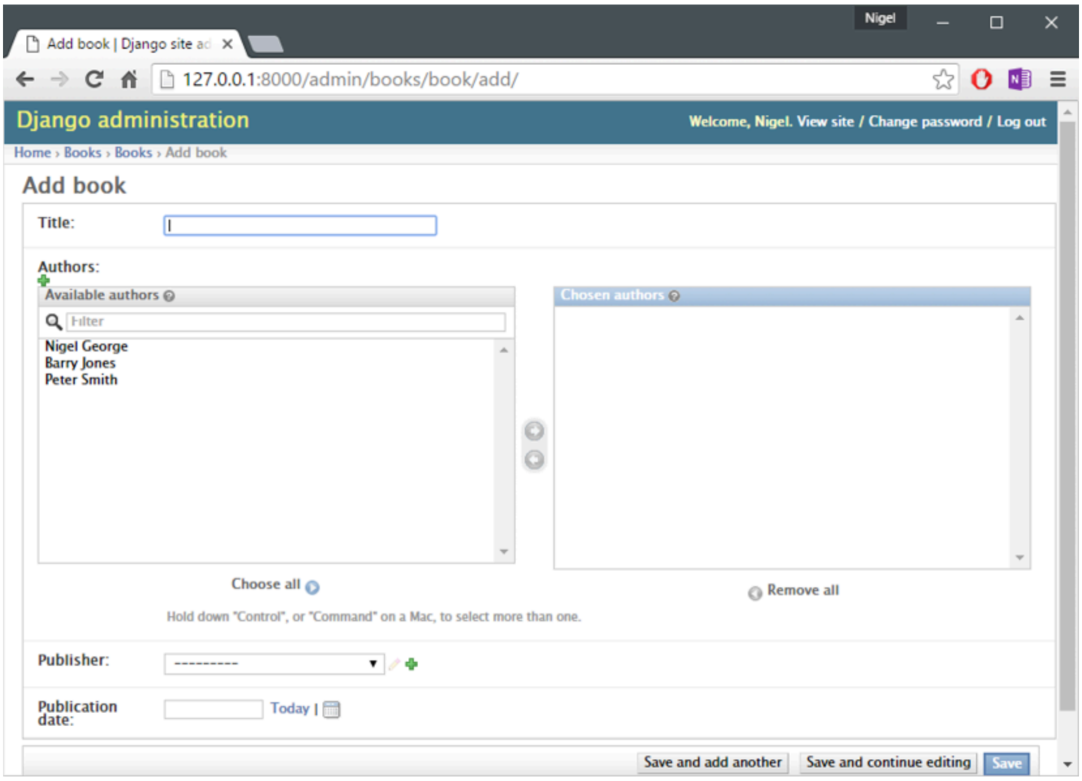


图 5-13：添加 `filter_horizontal` 后的图书编辑表单

如果多对多字段中的项目超过 10 个，我强烈建议使用 `filter_horizontal`。这个过滤器界面比简单的多项选择框容易使用多了。此外注意，`filter_horizontal` 可以指定多个字段，在元组中列出各个字段的名称即可。

`ModelAdmin` 子类也支持 `filter_vertical` 选项。它的作用与 `filter_horizontal` 完全一样，不过得到的 JavaScript 界面是纵向排列的两个选择框，而不是横向的。使用哪个全看个人喜好。

`filter_horizontal` 和 `filter_vertical` 只能用于多对多字段，不能用于外键字段。默认情况下，管理后台为外键字段显示一个简单的 `<select>` 菜单，不过与多对多字段一样，有时你不想费力气从下拉菜单中找出所需的对象。

例如，书籍数据库不断变大，包含几千个出版社，添加图书表单要花一些时间才能加载，因为要加载每个出版社记录，在 `<select>` 菜单中显示。

为了解决这个问题，可以使用 `raw_id_fields` 选项：

```
class BookAdmin(admin.ModelAdmin):
    list_display = ('title', 'publisher','publication_date')
    list_filter = ('publication_date',)
    date_hierarchy = 'publication_date'
    ordering = ('-publication_date',)
    filter_horizontal = ('authors',)
    raw_id_fields = ('publisher',)
```

这个选项的值是一个外键字段名称元组，各个字段在管理后台中显示为简单的文本输入框（`<input type="text">`），而不是 `<select>` 菜单，如图 5-14 所示。

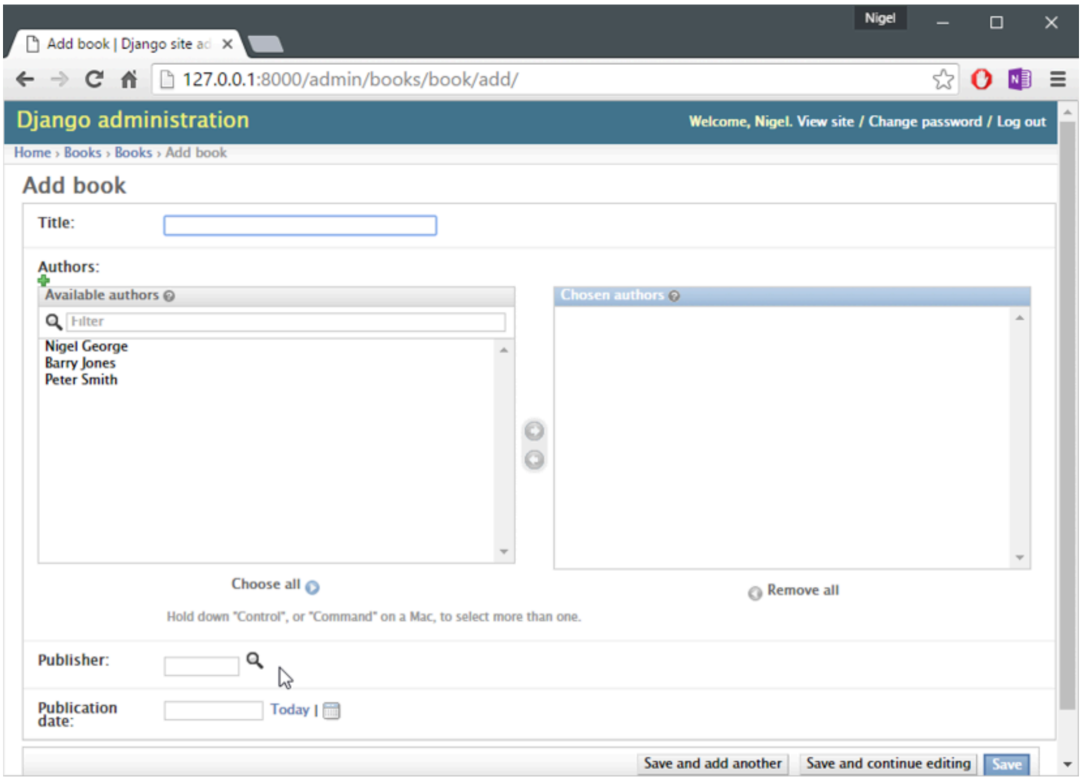


图 5-14：添加 `raw_id_fields` 后的图书编辑表单

在这个输入框中输入什么呢？数据库中出版社记录的 ID。鉴于人类通常不善于记忆数字，旁边还有一个放大镜图标，点击后弹出一个窗口，用于选择要添加的出版社。

5.6 用户、分组和权限

你是以超级用户身份登录的，有权限创建、编辑和删除任何对象。当然，不同的环境需要不同的权限系统，不是每个人都可以或应该是超级用户。Django 的管理后台也有权限系统，让你给特定用户赋予访问特定功能的权限。用户帐号应该是通用的、独立于管理界面，在外部仍可以使用，但是我们现在仍把他们当做管理员账户。

第 11 章将说明如何使用 Django 的身份认证系统管理全站的用户（即不仅是管理后台的用户）。用户和权限可以像其他对象一样在管理界面中编辑。本章前面介绍管理后台的“Users”和“Groups”部分时已经有所涉及。

正如你期望的，用户对象有一些标准的字段：用户名、密码、电子邮件和真实姓名。此外，还有一些字段用于定义允许用户在管理界面中做什么。首先是三个布尔值旗标：

- “Active”控制是否激活用户。如果未勾选，即便用户使用有效的密码也无法登录。
- “Staff status”控制是否允许用户登录管理界面（即是否把用户当做组织中的一员）。因为这个用户系统也用于控制面向公众的网站（即前台，参见第 11 章），所以这个旗标对公开用户和管理员是有区别的。
- “Superuser status”为用户赋予所有权限，可以在管理界面中添加、编辑和删除任何对象。如果勾选，用户的常规权限（即使没有）不再考虑。

“普通的”管理员，即已激活且不是超级用户，所具有的管理权限是一项项赋予的。可在管理界面中编辑的对象（如图书、作者和出版社）有三个权限：创建权限、编辑权限和删除权限。为用户赋予权限就是允许用户

执行相应的操作。新建的用户没有任何权限，如果需要特定权限，要由你赋予。

例如，可以为用户赋予添加和修改出版社的权限，但是不允许删除。注意，这些权限是针对每个模型的，而不是每个对象。因此，可以允许 John 修改任何一本书，但是不能允许他只能修改 Apress 出版的书。针对各个对象的权限更复杂，不在本书的讨论范围之内，不过 Django 的文档中有说明。

> **提醒**
>
> 编辑用户和权限的权限也由权限系统控制。如果为某人赋予编辑用户的权限，他就能编辑自己的权限——这可能不是你想要的行为！给用户赋予编辑用户的权限，其实就相当于把他变成超级用户。

用户还可以分组。一个分组中的所有成员都有那一组具有的全部权限。使用分组便于为多个用户赋予相同的权限。

5.7 何时以及为何使用管理界面

至此，你应该基本知道该如何使用 Django 的管理后台了。不过，我想说明一下何时以及为何使用管理后台，以及何时无需使用。

对想输入数据的非技术人员来说，Django 的管理后台特别有用；毕竟，这就是管理后台的目的。在 Django 诞生的新闻业中，一个在线功能（例如市政供水水质特别报道）的开发过程通常是这样的：

- 负责该项目的记者与一位开发者碰头，指出所需的数据。
- 开发者设计满足需求的 Django 模型，然后打开管理后台给记者看。
- 记者审查管理后台，及时指出缺少或多余的字段。开发者不断修改模型。
- 得到满意的模型之后，记者开始在管理后台中输入数据。与此同时，程序员可以集中精力开发面向公众的视图/模板（即开发过程中有趣的部分）。

也就是说，Django 的管理界面为内容制作人员和程序员都提供了便利的工具。除了输入数据之外，管理后台还有很多用处：

- 审查数据模型：定义几个模型之后，可以在管理界面中查看，输入一些虚拟数据。有时，在这个过程中能够发现数据建模等问题。
- 管理从别处得到的数据：对依靠外部源（例如用户或 Web 爬虫）提供数据的应用程序来说，通过管理后台便于审查或编辑数据。你可以把管理后台看做数据库命令行工具的另一个版本，虽然不那么强大，但是足够便利。
- 临时的数据管理应用：你可以使用管理后台构建一个特别轻量级的数据管理应用，例如记录花销。如果只构建给自己用的功能，而不面向公众，管理后台能节省很多时间。在这个意义上，管理后台相当于增强版关系型电子表格。

然而，管理后台不是万能的。不应该把它当做数据的公开界面，它也不具有复杂的排序和搜索功能。前面说过，管理后台是供授信的网站管理员使用的。唯有记住这一点，才能有效利用管理后台。

5.8 接下来

至此，我们创建了几个模型，也配置了一流的编辑界面。下一章将换个话题，进入 Web 开发的实质阶段：创建和处理表单。

第 6 章 Django 表单

HTML 表单是交互式网站的基本组成部分，从 Google 网站中简单的搜索框，到无处不在的博客评论提交表单，再到复杂的数据输入界面，都能见到表单的身影。

本章说明如何使用 Django 访问、验证和处理用户提交的表单数据。在这个过程中，我们将介绍 `HttpRequest` 和 `Form` 对象。

6.1 从请求对象中获取数据

第 2 章首次介绍视图函数时提到过 `HttpRequest` 对象，但是没有细讲。还记得吗，每个视图函数的第一个参数都是一个 `HttpRequest` 对象，如下面的 `hello()` 视图所示：

```
from django.http import HttpResponse

def hello(request):
    return HttpResponse("Hello world")
```

`HttpRequest` 对象，如这里的 `request` 参数，有一些有用的属性和方法，你应该有所了解，这样才知道有什么可用。执行视图函数时，可以使用这些属性获取关于当前请求（即用户在 Web 浏览器中访问 Django 驱动的网站中的某个页面）的信息。

6.1.1 关于 URL 的信息

`HttpRequest` 对象中有一些关于当前所请求 URL 的信息（表 6-1）。

表 6-1：`HttpRequest` 对象的方法和属性

属性/方法	说明	示例
`request.path`	完整的路径，不含域名，但是包含前导斜线	“/hello/”
`request.get_host()`	主机名（即通常所说的“域名”）	“127.0.0.1:8000”或“www.example.com”
`request.get_full_path()`	包含查询字符串（如果有的话）的路径	“/hello/?print=true”
`request.is_secure()`	通过 HTTPS 访问时为 `True`，否则为 `False`	`True` 或 `False`

在视图中一定要使用这些属性或方法，不能硬编码 URL。这样写出的代码更灵活，便于在不同的地方复用。下面举个简单的例子：

```
# 不好
def current_url_view_bad(request):
    return HttpResponse("Welcome to the page at /current/")
```

```python
# 好
def current_url_view_good(request):
    return HttpResponse("Welcome to the page at %s" % request.path)
```

6.1.2 关于请求的其他信息

request.META 的值是一个 Python 字典，包含请求的所有 HTTP 首部，例如用户的 IP 地址和*用户代理*（user agent，通常是 Web 浏览器的名称和版本）。注意，具体包含哪些首部取决于用户发送了什么首部，以及 Web 服务器返回了什么首部。这个字典中常见的几个键有：

- HTTP_REFERER：入站前的 URL（可能没有）。（注意，要使用错误的拼写，即 REFERER。）
- HTTP_USER_AGENT：浏览器的用户代理（可能没有）。例如："Mozilla/5.0 (X11; U; Linux i686; fr-FR; rv:1.8.1.17) Gecko/20080829 Firefox/2.0.0.17"。
- REMOTE_ADDR：客户端的 IP 地址，例如 "12.345.67.89"。（如果请求经由代理，这个首部的值可能是一组 IP 地址，以逗号分隔，例如 "12.345.67.89,23.456.78.90"。）

注意，因为 request.META 是个普通的 Python 字典，所以尝试访问不存在的键时，抛出 KeyError 异常。（HTTP 首部是外部数据，即由用户的浏览器提交，因此不能完全相信，当某个首部为空或不存在时，应该让应用程序优雅失败。）为了处理未定义的键，应该使用 try/except 子句，或者 get() 方法：

```python
# 不好
def ua_display_bad(request):
    ua = request.META['HTTP_USER_AGENT']  # 可能抛出 KeyError
    return HttpResponse("Your browser is %s" % ua)

# 好（版本 1）
def ua_display_good1(request):
    try:
        ua = request.META['HTTP_USER_AGENT']
    except KeyError:
        ua = 'unknown'
    return HttpResponse("Your browser is %s" % ua)

# 好（版本 2）
def ua_display_good2(request):
    ua = request.META.get('HTTP_USER_AGENT', 'unknown')
    return HttpResponse("Your browser is %s" % ua)
```

建议你编写一个简单的视图，显示 request.META 中的所有信息，以便查阅。下面是个示例：

```python
def display_meta(request):
    values = request.META.items()
    values.sort()
    html = []
    for k, v in values:
        html.append('<tr><td>%s</td><td>%s</td></tr>' % (k, v))
    return HttpResponse('<table>%s</table>' % '\n'.join(html))
```

查看请求对象中有哪些信息的另一个好办法是仔细分析 Django 的错误页面，里面有很多有用的信息，包括全部 HTTP 首部和其他请求信息（如 request.path）。

6.1.3 关于提交数据的信息

除了关于请求的基本元数据之外，`HttpRequest` 对象还有两个属性包含用户提交的信息：`request.GET` 和 `request.POST`。这两个属性的值都是类似字典的对象，分别用于获取 `GET` 和 `POST` 数据。`POST` 数据一般由 HTML 表单提交，而 `GET` 数据既可以来自表单，也可以来自页面 URL 中的查询字符串。

> **类似字典的对象**
>
> 我们说 `request.GET` 和 `request.POST` 是"类似字典的对象"，意思是它们的行为与标准的 Python 字典相似，但是不完全一样。例如，`request.GET` 和 `request.POST` 都有 `get()`、`keys()` 和 `values()` 等方法，而且可以使用 `for key in request.GET` 迭代键。那么，为什么不干脆使用字典呢？因为 `request.GET` 和 `request.POST` 都有常规的字典没有的额外方法，稍后会做介绍。你可能见过类似的表述，例如"类似文件的对象"，这种对象有些基本的方法，如 `read()`，行为与"真正的"文件对象相似。

6.2 一个简单的表单处理示例

我们继续以图书、作者和出版社为例。下面创建一个简单的视图，让用户通过书名搜索数据库中的图书。一般来说，表单分为两部分：用户界面 HTML 和处理提交数据的后端视图代码。前半部分很简单，只需编写一个视图，显示搜索表单：

```
from django.shortcuts import render

def search_form(request):
    return render(request, 'search_form.html')
```

读过第 3 章我们知道，视图可以放在 Python 路径中的任何位置。但是，为了便于增强，我们把它放在 `books/views.py` 文件中。对应的 `search_form.html` 模板如下所示：

```
<html>
<head>
    <title>Search</title>
</head>
<body>
    <form action="/search/" method="get">
        <input type="text" name="q">
        <input type="submit" value="Search">
    </form>
</body>
</html>
```

把这个文件保存在第 3 章创建的 `mysite/templates` 目录中。此外，也可以新建 `books/templates` 文件夹。此时应该把设置文件的 `'APP_DIRS'` 设为 `True`。`urls.py` 文件中的 URL 模式如下：

```
from books import views

urlpatterns = [
    # ...
    url(r'^search-form/$', views.search_form),
    # ...
]
```

（注意，我们直接导入 views 模块，而不是 from mysite.views import search_form，因为前者更简单。第 7 章将详细说明这种导入方式。）此时，启动开发服务器，访问 http://127.0.0.1:8000/search-form/，你会看到搜索界面。就这么简单！不过，提交表单时会显示 Django 404 错误。这是因为表单指向的 URL /search/ 还未实现。下面再编写一个视图函数，修正这个问题：

```
# urls.py

urlpatterns = [
    # ...
    url(r'^search-form/$', views.search_form),
    url(r'^search/$', views.search),
    # ...
]

# books/views.py

from django.http import HttpResponse

# ...

def search(request):
    if 'q' in request.GET:
        message = 'You searched for: %r' % request.GET['q']
    else:
        message = 'You submitted an empty form.'
    return HttpResponse(message)
```

目前，这个视图只是显示用户搜索的词。由此，我们可以确认数据正确提交给 Django 了，也可以了解搜索的词是如何在系统中传递的。简单来说：

1. HTML 表单定义一个变量 q。提交表单后，q 的值通过 GET 请求（method="get"）传给 /search/。
2. 处理 /search/ 的 Django 视图（search()）通过 request.GET 访问 q 的值。

特别注意，我们明确检查了 request.GET 中有没有 'q'。6.1.2 节说过，不应该相信用户提交的任何数据，甚至可以假设用户根本没有提交任何数据。如果不做这一项检查，提交空表单会导致视图抛出 KeyError：

```
# 不好
def bad_search(request):
    # 如果未提交 'q'，下一行抛出 KeyError
    message = 'You searched for: %r' % request.GET['q']
    return HttpResponse(message)
```

6.2.1 查询字符串参数

GET 数据通过查询字符串传递（例如 /search/?q=django），因此可以使用 request.GET 获取查询字符串参数。第 2 章介绍 Django 的 URL 配置系统时，我比较了 Django 的精美 URL 和传统的 PHP/Java URL，如 /time/plus?hours=3，我说我会在第 6 章说明怎么得到后一种 URL。现在你知道如何在视图中访问查询字符串参数了（如这里的 hours=3），即使用 request.GET。

POST 数据的访问方式与 GET 数据类似，只需把 request.GET 换成 request.POST。那么，GET 和 POST 之间有什么区别呢？如果提交表单只是为了“获取”数据，使用 GET。如果提交表单有副作用，例如修改数据、发送电子邮件，或者是显示数据之外的操作，使用 POST。在这个搜索图书的示例中，我们使用的是 GET，因为搜索

操作没有修改服务器中的任何数据。（如果想进一步了解 GET 和 POST，请访问 w3.org 网站。）现在，我们已经确认 request.GET 能正确传进来了，下面在图书数据库中执行用户提交的搜索查询（还是在 views.py 文件中）：

```
from django.http import HttpResponse
from django.shortcuts import render
from books.models import Book

def search(request):
    if 'q' in request.GET and request.GET['q']:
        q = request.GET['q']
        books = Book.objects.filter(title__icontains=q)
        return render(request, 'search_results.html',
                      {'books': books, 'query': q})
    else:
        return HttpResponse('Please submit a search term.')
```

在这段代码中，有几处要注意：

- 除了检查 request.GET 中有没有 'q' 之外，我们还确保 request.GET['q'] 不是空值，然后再把查询传给数据库。
- 我们使用 Book.objects.filter(title__icontains=q) 在图书表中查找所有书名中包含查询词条的书。icontains 是一种查找类型（参见第 4 章和附录 B），这个语句基本上相当于“获取所有书名中包含 q 的书，而且不区分大小写”。

这种搜索方式十分简单。不建议在大型生产数据库中使用 icontains 查询，因为速度可能很慢。（在实际运用中，你可能想使用某种自定义的搜索系统。你可以搜索一下开源的全文搜索引擎。）

我们把一组 Book 对象 books 传给模板。search_results.html 文件可以像这样编写：

```
<html>
    <head>
    <title>Book Search</title>
    </head>
    <body>
        <p>You searched for: <strong>{{ query }}</strong></p>

        {% if books %}
            <p>Found {{ books|length }} book{{ books|pluralize }}.</p>
            <ul>
                {% for book in books %}
                <li>{{ book.title }}</li>
                {% endfor %}
            </ul>
        {% else %}
            <p>No books matched your search criteria.</p>
        {% endif %}
    </body>
</html>
```

注意，这里用到了 pluralize 模板过滤器，它会根据找到的图书数量输出正确的单复数。

6.3 改进这个简单的表单处理示例

与前面几章一样，我展示的是最简可用的方法。现在，我要指出一些问题，并说明如何改进。首先，search() 视图对空查询的处理不完美，我们只是显示“Please submit a search term.”消息，用户必须点击浏览器的后退按钮。

这样做不友好，而且显得不专业，这样的实现会再次经由 Django 处理。遇到这种情况时，重新显示表单，并且在上部显示错误更好，这样用户可以立即再试一次。为此，最简单的方法是再次渲染模板，如下所示：

```
from django.http import HttpResponse
from django.shortcuts import render
from books.models import Book

def search_form(request):
    return render(request, 'search_form.html')

def search(request):
    if 'q' in request.GET and request.GET['q']:
        q = request.GET['q']
        books = Book.objects.filter(title__icontains=q)
        return render(request, 'search_results.html',
                      {'books': books, 'query': q})
    else:
        return render(request, 'search_form.html',
                      {'error': True})
```

（注意，我还列出了 search_form()，以便让你对比两个视图。）我们改进了 search() 视图，在查询词条为空时再次渲染 search_form.html 模板。因为需要在模板中显示错误消息，所以我们还传入了一个模板变量。现在，我们可以编辑 search_form.html，检查 error 变量：

```
<html>
<head>
    <title>Search</title>
</head>
<body>
    {% if error %}
        <p style="color: red;">Please submit a search term.</p>
    {% endif %}
    <form action="/search/" method="get">
        <input type="text" name="q">
        <input type="submit" value="Search">
    </form>
</body>
</html>
```

我们可以继续使用 search_form() 视图，因为它没有把 error 变量传给模板，所以不会显示错误消息。这样修改之后，我们的应用程序变得更好了，但是我们不禁要问：真的有必要专门编写一个 search_form() 视图吗？

按照现在的处理方式，对 /search/ 的请求（没有 GET 参数）会显示空的表单（没有错误）。只要我们修改 search() 视图，在直接访问 /search/ 时（没有 GET 参数）隐藏错误消息，就可以把 search_form() 视图及对应的 URL 模式删除：

```
def search(request):
    error = False
    if 'q' in request.GET:
        q = request.GET['q']
        if not q:
            error = True
        else:
            books = Book.objects.filter(title__icontains=q)
            return render(request, 'search_results.html',
                          {'books': books, 'query': q})
    return render(request, 'search_form.html',
                  {'error': error})
```

这样修改之后，当用户直接访问 /search/ 时，看到的是没有错误的搜索表单；如果用户提交 'q' 为空值的表单，看到的是带有错误消息的搜索表单；如果用户提交 'q' 不为空值的表单，看到的是搜索结果。

最后，我们要删除一些重复代码。现在，我们把两个视图和 URL 合并为一个了，/search/ URL 既能显示搜索表单，也能显示搜索结果，那么 search_form.html 中的 HTML <form> 不再需要硬编码 URL 了。我们要把

```
<form action="/search/" method="get">
```

改为

```
<form action="" method="get">
```

action="" 的意思是，“把表单提交到与当前页面相同的 URL”。这样修改之后，如果想把 search() 视图放到别的 URL 上，不用再修改 action 属性。

6.4 简单的验证

这个搜索示例还相当简单，尤其是在数据验证方面。我们只是检查搜索词条是否为空。很多 HTML 表单采用的验证措施比确认值不为空复杂得多。我们经常在网站中见到这样的错误消息：

- “请输入有效的电子邮件地址。‘foo’不是电子邮件地址。”
- “请输入有效的五位数美国邮编。‘123’不是邮编。”
- “请输入有效的日期，格式为 YYYY-MM-DD。”
- “请输入至少有 8 个字符的密码，而且至少有一个数字。”

下面我们来调整一下 search() 视图，让它验证搜索词条的长度，确保小于或等于 20 个字符。（对这个示例来说，我们假设超过这一限制的词条搜索起来太慢。）但是怎么实现呢？

最简单的做法可能是直接在视图中实现相关逻辑，如下所示：

```
def search(request):
    error = False
    if 'q' in request.GET:
        q = request.GET['q']
        if not q:
            error = True
        elif len(q) > 20:
            error = True
        else:
```

```
            books = Book.objects.filter(title__icontains=q)
            return render(request, 'search_results.html',
                          {'books': books, 'query': q})
    return render(request, 'search_form.html',
        {'error': error})
```

现在，如果搜索词条超过 20 个字符，无法搜索，你会看到一个错误消息。但是，search_form.html 模板目前显示的错误消息是“Please submit a search term.”，因此我们要做修改，在两种情况下显示不同的消息：

```
<html>
<head>
    <title>Search</title>
</head>
<body>
    {% if error %}
        <p style="color: red;">
            Please submit a search term 20 characters or shorter.
        </p>
    {% endif %}

    <form action="" method="get">
        <input type="text" name="q">
        <input type="submit" value="Search">
    </form>
</body>
</html>
```

但是，现在还有问题：所有错误都显示同一个错误消息，让人不解。为什么提交空值时显示的错误消息要提到 20 个字符限制呢？

错误消息应该具体、无歧义、易于理解。问题的根源是，我们把 error 变量的值设成了布尔值，其实应该设为一个错误消息字符串列表。修正的方法如下：

```
def search(request):
    errors = []
    if 'q' in request.GET:
        q = request.GET['q']
        if not q:
            errors.append('Enter a search term.')
        elif len(q) > 20:
            errors.append('Please enter at most 20 characters.')
        else:
            books = Book.objects.filter(title__icontains=q)
            return render(request, 'search_results.html',
                          {'books': books, 'query': q})
    return render(request, 'search_form.html',
                  {'errors': errors})
```

然后，还要稍微修改一下 search_form.html 模板，把布尔值 error 改成列表：

```
<html>
<head>
    <title>Search</title>
</head>
```

```
<body>
    {% if errors %}
        <ul>
            {% for error in errors %}
            <li>{{ error }}</li>
            {% endfor %}
        </ul>
    {% endif %}
    <form action="" method="get">
        <input type="text" name="q">
        <input type="submit" value="Search">
    </form>
</body>
</html>
```

6.5 创建一个联系表单

虽然我们不断改进图书搜索表单，但是它的功能依旧很简单，只有一个字段，即 'q'。表单变复杂之后，我们要重复上述步骤，处理每个表单字段。在这个过程中容易引入大量混乱，而且人为出错的可能性很大。幸运的是，Django 的开发者想到了这一点，在 Django 中内建了高层级的库，能够处理表单和验证相关的任务。

6.5.1 第一个表单类

Django 自带了一个表单库，django.forms，它能处理本章所述的多数问题，从显示 HTML 表单到验证，都能胜任。下面我们使用这个 Django 表单框架为应用程序构建一个联系表单。

这个表单框架的主要用法是为要处理的每个 HTML 表单定义一个 Form 类。这里只有一个表单，因此只需定义一个 Form 类。这个类可以放在任意位置，例如直接放在 views.py 文件中，不过社区的约定是，把 Form 类放在单独的 forms.py 文件中。

在 views.py 文件所在的目录（mysite）中创建这个文件，然后输入下述内容：

```
from django import forms

class ContactForm(forms.Form):
    subject = forms.CharField()
    email = forms.EmailField(required=False)
    message = forms.CharField()
```

这段代码易于理解，而且与 Django 模型的句法相似。表单中的各个字段使用相应的 Field 类表示（这里只用到了 CharField 和 EmailField），定义为 Form 类的属性。字段默认是必填的，因此为了把 email 设为选填，我们指定了 required=False。下面我们在 Python 交互式解释器中看看这个类能做什么。首先，它能把自己显示为 HTML：

```
>>> from mysite.forms import ContactForm
>>> f = ContactForm()
>>> print(f)
<tr><th><label for="id_subject">Subject:</label></th><td><input type="text" name="sub\
ject" id="id_subject" /></td></tr><tr><th><label for="id_email">Email:</label></th><t\
d><input type="text" name="email" id="id_email" /></td></tr><tr><th><label for="id_me\
ssage">Message:</label></th><td><input type="text" name="message" id="id_message" /><\
```

```
/td></tr>
```

为了可访问性，Django 会为各个字段添加标注（`<label>` 标签）。Django 会尽力把默认行为做到最好。默认的输出使用 HTML 表格，不过还有几个其他的内置输出格式：

```
>>> print(f.as_ul())
<li><label for="id_subject">Subject:</label>
<input type="text" name="subject" id="id_subject" /></li><li><label for="id_email">Em\
ail:</label> <input type="text" name="email" id="id_email" /></li><li><label for="id_\
message">Message:</label><input type="text" name="message" id="id_message"/></li>

>>> print(f.as_p())
<p><label for="id_subject">Subject:</label><input type="text" name="subject" id="id_s\
ubject"/></p><p><label for="id_email">Email:</label> <input type="text" name="email" \
id="id_email" /></p><p><label for="id_message">Message:</label><input type="text" nam\
e="message" id="id_message"/></p>
```

注意，输出中没有 `<table>`、`<ul>` 和 `<form>` 起始和结束标签，所以你可以根据需要添加字段，或者做其他定制。以上是显示整个表单的方式，此外还可以只显示某个字段的 HTML：

```
>>> print(f['subject'])
<input id="id_subject" name="subject" type="text" />
>>> print f['message']
<input id="id_message" name="message" type="text" />
```

`Form` 对象能做的第二件事是验证数据。为此，创建一个 `Form` 对象，传入一个数据字典，把字段名映射到数据上：

```
>>> f = ContactForm({'subject': 'Hello', 'email': 'adrian@example.com', 'message': 'Nice
site!'})
```

为 `Form` 实例关联数据后，得到了一个“受约束的”表单：

```
>>> f.is_bound
True
```

然后，在受约束的表单对象上调用 `is_valid()` 方法，检查数据是否有效。前面为各个字段传入的值都是有效的，因此这个表单对象是完全有效的：

```
>>> f.is_valid()
True
```

如果不传入 `email` 字段，仍然是有效的，因为我们为那个字段指定了 `required=False`：

```
>>> f = ContactForm({'subject': 'Hello', 'message': 'Nice site!'})
>>> f.is_valid()
True
```

但是，如果不提供 `subject` 或 `message` 字段，表单对象就变成无效的了：

```
>>> f = ContactForm({'subject': 'Hello'})
>>> f.is_valid()
False
>>> f = ContactForm({'subject': 'Hello', 'message': ''})
>>> f.is_valid()
```

```
False
```

我们可以深入查看各个字段的错误消息：

```
>>> f = ContactForm({'subject': 'Hello', 'message': ''})
>>> f['message'].errors
['This field is required.']
>>> f['subject'].errors
[]
>>> f['email'].errors
[]
```

每个受约束的表单实例都有一个 errors 属性，它的值是一个字典，把字段名称映射到错误消息列表上：

```
>>> f = ContactForm({'subject': 'Hello', 'message': ''})
>>> f.errors
{'message': ['This field is required.']}
```

最后，具有有效数据的表单实例有个 cleaned_data 属性，它的值是一个字典，存储着"清理后的"提交数据。Django 的表单框架不仅验证数据，还会清理数据，把值转换成合适的 Python 类型：

```
>>> f = ContactForm({'subject': 'Hello', 'email': 'adrian@example.com',
'message': 'Nice site!'})
>>> f.is_valid() True
>>> f.cleaned_data
{'message': 'Nice site!', 'email': 'adrian@example.com', 'subject':
'Hello'}
```

这个联系表单只处理字符串，经"清理"后得到字符串对象。然而，如果使用 IntegerField 或 DateField，表单框架会确保清理后的数据使用正确的 Python 整数或 datetime.date 对象。

6.6 在视图中使用表单对象

如果不向用户显示，这个联系表单没有什么用。为此，首先要更新 mysite/views：

```
# views.py

from django.shortcuts import render
from mysite.forms import ContactForm
from django.http import HttpResponseRedirect
from django.core.mail import send_mail

# ...

def contact(request):
    if request.method == 'POST':
        form = ContactForm(request.POST)
        if form.is_valid():
            cd = form.cleaned_data
            send_mail(
                cd['subject'],
                cd['message'],
                cd.get('email', 'noreply@example.com'),
```

```
                ['siteowner@example.com'],
            )
            return HttpResponseRedirect('/contact/thanks/')
    else:
        form = ContactForm()
    return render(request, 'contact_form.html', {'form': form})
```

接下来，要创建联系表单（保存在 mysite/templates 文件夹里）：

```
# contact_form.html

<html>
<head>
    <title>Contact us</title>
</head>
<body>
    <h1>Contact us</h1>

    {% if form.errors %}
        <p style="color: red;">
            Please correct the error{{ form.errors|pluralize }} below.
        </p>
    {% endif %}

    <form action="" method="post">
        <table>
            {{ form.as_table }}
        </table>
        {% csrf_token %}
        <input type="submit" value="Submit">
    </form>
</body>
</html>
```

最后，要修改 urls.py 文件，在 /contact/ URL 上显示联系表单：

```
# ...
from mysite.views import hello, current_datetime, hours_ahead, contact

urlpatterns = [

    # ...

    url(r'^contact/$', contact),
]
```

这个表单使用 POST 处理（要修改数据），因此要关心*跨站请求伪造*（Cross Site Request Forgery，CSRF）。幸好，我们无需太过担心，因为 Django 提供了非常易用的防护系统。简单来说，所有通过 POST 指向内部 URL 的表单都应该使用 {% csrf_token %} 模板标签。关于这个标签的详情，参见第 19 章。

在本地运行一下这个表单，不填写任何字段、填写无效的电子邮件地址，再填写有效的数据，分别提交试一下。（当然，如果没有配置邮件服务器，调用 send_mail() 时会抛出 ConnectionRefusedError。）

6.7 改变字段的渲染方式

在本地渲染这个表单后，你注意到的第一件事可能是 message 字段显示为 <input type="text">，而它应该为 <textarea>。这个问题可以通过设定字段的 widget 参数修正：

```
from django import forms

class ContactForm(forms.Form):
    subject = forms.CharField()
    email = forms.EmailField(required=False)
    message = forms.CharField(widget=forms.Textarea)
```

表单框架把各个字段的表现逻辑交给 widget 参数负责。每种字段的 widget 参数都有默认值，不过可以轻易覆盖，或者自定义。Field 类表示验证逻辑，而 widget 表示表现逻辑。

6.8 设定最大长度

最常见的验证需求是检查字段中的值是否为特定的长度。保险起见，我们应该改进 ContactForm，限制 subject 的值在 100 个字符以内。为此，只需为 CharField 指定 max_length 参数，如下所示：

```
from django import forms

class ContactForm(forms.Form):
    subject = forms.CharField(max_length=100)
    email = forms.EmailField(required=False)
    message = forms.CharField(widget=forms.Textarea)
```

此外，还有个 min_length 参数。

6.9 设定初始值

下面继续改进这个表单，为 subject 字段添加一个初始值："I love your site!"（提供一些建议总是好的。）为此，创建 Form 实例时可以提供 initial 参数：

```
def contact(request):
    if request.method == 'POST':
        form = ContactForm(request.POST)
        if form.is_valid():
            cd = form.cleaned_data
            send_mail(
                cd['subject'],
                cd['message'],
                cd.get('email', 'noreply@example.com'),
                ['siteowner@example.com'],
            )
            return HttpResponseRedirect('/contact/thanks/')
    else:
        form = ContactForm(
            initial={'subject': 'I love your site!'}
        )
    return render(request, 'contact_form.html', {'form':form})
```

现在，subject 字段中会显示一句赞扬的话。注意，传递初始值与传递绑定表单的数据不同，二者之间最大的区别是，传递初始数据时，表单不受约束，因此不会产生任何错误消息。

6.10 自定义验证规则

假如我们推出了反馈表单，电子邮件纷至沓来。这就引出一个问题：有些邮件可能只有一两个词，不知所云。因此，我们决定采取一个新的验证措施：建议写四个词以上。

在 Django 表单中使用自定义的验证有多种方式。如果验证规则要不断复用，可以自定义一个字段类型。不过，多数自定义的验证都是一次性的，可以直接写在 Form 类中。我们想对 message 字段做额外的验证，因此在 Form 类中添加一个 clean_message() 方法：

```
from django import forms

class ContactForm(forms.Form):
    subject = forms.CharField(max_length=100)
    email = forms.EmailField(required=False)
    message = forms.CharField(widget=forms.Textarea)

    def clean_message(self):
        message = self.cleaned_data['message']
        num_words = len(message.split())
        if num_words < 4:
            raise forms.ValidationError("Not enough words!")
        return message
```

Django 的表单系统会自动查找名称以 clean_ 开头、以字段名结尾的方法。如果存在这样的方法，在验证过程中调用。这里，clean_message() 方法会在指定字段的默认验证逻辑（这个 CharField 是必填的）执行完毕后调用。

因为字段数据经过部分处理了，所以从 self.cleaned_data 中获取。此外，我们无需检查值是否存在，或者不为空，默认的验证逻辑已经检查过了。我们的处理方式很简单，只是使用 len() 和 split() 计算单词的数量。如果用户输入的词数过少，抛出 forms.ValidationError。

这个异常中指定的字符串会出现在错误消息列表中显示给用户。注意，方法的最后一定要显式返回那个字段清理后的值。这样才能在自定义的验证方法中修改那个值（或者转换成其他 Python 类型）。如果没有 return 语句，返回的是 None，如此一来原来的值就丢失了。

6.11 指定标注

在 Django 自动生成的表单 HTML 中，默认的标注是把下划线替换成空格，并把首字母变成大写，例如，email 字段的标注是“Email”。（听着很熟悉吧，这与 Django 的模型为字段生成 verbose_name 的值使用的算法相同。我们在第 4 章介绍过。）与 Django 的模型一样，字段的标注是可以自定义的。方法很简单，指定 label 参数即可，如下所示：

```
class ContactForm(forms.Form):
    subject = forms.CharField(max_length=100)
    email = forms.EmailField(required=False, label='Your e-mail address')
    message = forms.CharField(widget=forms.Textarea)
```

6.12 自定义表单的外观

contact_form.html 模板使用 {{ form.as_table }} 显示表单，不过我们可以通过其他方式显示表单，进一步控制外观。自定义表单外观最简便的方法是使用 CSS。

错误列表尤其可以做些外观上的改进。自动生成的错误列表使用 <ul class="errorlist">，因此可以使用 CSS 美化。下述 CSS 把错误列表突出显示出来：

```
<style type="text/css">
    ul.errorlist {
        margin: 0;
        padding: 0;
    }
    .errorlist li {
        background-color: red;
        color: white;
        display: block;
        font-size: 10px;
        margin: 0 0 3px;
        padding: 4px 5px;
    }
</style>
```

虽然可以让 Django 为我们生成表单的 HTML，但是很多时候我们并不想用默认的渲染方式。{{ form.as_table }} 等是有用的快捷方式，在开发应用程序的过程中能节省时间，但是表单的显示方式是可以定制的，而且大多数时候都在模板中——你肯定会这么做的。

在模板中，每个字段使用的控件（<input type="text">、<select>、<textarea>，等等）可以使用 {{ form.fieldname }} 单独渲染，而各个字段上的错误可以通过 {{ form.fieldname.errors }} 获取。

了解这些之后，我们可以使用下述代码自定义联系表单的模板：

```
<html>
<head>
    <title>Contact us</title>
</head>
<body>
    <h1>Contact us</h1>

    {% if form.errors %}
        <p style="color: red;">
            Please correct the error{{ form.errors|pluralize }} below.
        </p>
    {% endif %}

    <form action="" method="post">
        <div class="field">
            {{ form.subject.errors }}
            <label for="id_subject">Subject:</label>
            {{ form.subject }}
        </div>
        <div class="field">
            {{ form.email.errors }}
```

```
            <label for="id_email">Your e-mail address:</label>
            {{ form.email }}
        </div>
        <div class="field">
            {{ form.message.errors }}
            <label for="id_message">Message:</label>
            {{ form.message }}
        </div>
        {% csrf_token %}
        <input type="submit" value="Submit">
    </form>
</body>
</html>
```

有错误时，{{ form.message.errors }} 显示一个 <ul class="errorlist"> 元素；字段有效时（或者表单未受约束时），显示一个空白字符串。我们可以把 form.message.errors 视作布尔值，也可以视作可迭代的列表。例如：

```
<div class="field{% if form.message.errors %} errors{% endif %}">
    {% if form.message.errors %}
        <ul>
        {% for error in form.message.errors %}
            <li><strong>{{ error }}</strong></li>
        {% endfor %}
        </ul>
    {% endif %}
    <label for="id_message">Message:</label>
    {{ form.message }}
</div>
```

有验证错误时，上述代码在 <div> 容器中添加 errors 类，并且在一个无序列表中显示错误。

6.13 接下来

至此，本书对基础知识（也叫“核心课程”）的介绍结束了。在接下来的一部分，即第 7 章到第 13 章，我们将讨论更高级的 Django 用法，例如如何部署 Django 应用程序（第 13 章）。读完前六章，你应该可以着手编写自己的 Django 项目了。本书剩下的内容为你提供补充知识。在第 7 章，我们将回过头来深入探讨视图和 URL 配置（第 2 章简单介绍过）。

第 7 章 高级视图和 URL 配置

第 2 章简单介绍了 Django 的视图函数和 URL 配置，本章深入更多细节，学习这两部分的高级功能。

7.1 URL 配置小技巧

URL 配置没什么特殊的，与 Django 的其他部分一样，只不过是普通的 Python 代码。鉴于此，我们可以利用一些 Python 技巧，充分发挥 URL 配置的作用，详情参见接下来的几节。

7.1.1 简化导入函数的方式

我们来看一下第 2 章中的一个 URL 配置：

```
from django.conf.urls import include, url
from django.contrib import admin
from mysite.views import hello, current_datetime, hours_ahead

urlpatterns = [
    url(r'^admin/', include(admin.site.urls)),
    url(r'^hello/$', hello),
    url(r'^time/$', current_datetime),
    url(r'^time/plus/(\d{1,2})/$', hours_ahead),
]
```

第 2 章说过，URL 配置中的每个条目都有对应的视图函数，通过函数对象直接传入。因此，要在 URL 配置模块的顶部导入视图函数。

但是，随着 Django 应用程序越变越复杂，URL 配置的内容会越变越多，一个一个地导入视图函数就显得繁琐。（每个视图函数都要导入，导入语句会变得越来越长。）

为了避免这种麻烦，我们可以导入 `views` 模块自身。下述 URL 配置与前面的效果一样：

```
from django.conf.urls import include, url
from . import views

urlpatterns = [
    url(r'^hello/$', views.hello),
    url(r'^time/$', views.current_datetime),
    url(r'^time/plus/(d{1,2})/$', views.hours_ahead),
]
```

7.1.2 在调试模式下提供特殊的 URL

说到动态构建 `urlpatterns`，你可能想利用这个技巧在 Django 的调试模式下调整 URL 配置的行为。为此，只需在运行时检查 `DEBUG` 设置，如下所示：

```
from django.conf import settings
```

```
from django.conf.urls import url
from . import views

urlpatterns = [
    url(r'^$', views.homepage),
    url(r'^(\d{4})/([a-z]{3})/$', views.archive_month),
]

if settings.DEBUG:
    urlpatterns += [url(r'^debuginfo/$', views.debug),]
```

这里，仅当 DEBUG 的值为 True 时，/debuginfo/ URL 才可以访问。

7.1.3 具名分组

上述示例没有使用具名的正则表达式分组（通过括号实现）捕获 URL 中的片段，而是通过位置参数把片段传给视图。

在高级用法中，可以使用具名的正则表达式分组捕获 URL 片段，通过关键字参数把片段传给视图。

在 Python 正则表达式中，具名分组的句法是 (?P<name>pattern)，其中 name 是分组的名称，pattern 是要匹配的模式。

假如我们的图书网站中有一些书评，而我们想检索特定日期或日期范围内的书评。

下面是 URL 配置：

```
from django.conf.urls import url
from . import views

urlpatterns = [
    url(r'^reviews/2003/$', views.special_case_2003),
    url(r'^reviews/([0-9]{4})/$', views.year_archive),
    url(r'^reviews/([0-9]{4})/([0-9]{2})/$', views.month_archive),
    url(r'^reviews/([0-9]{4})/([0-9]{2})/([0-9]+)/$', views.review_detail),
]
```

注意：

- 若想捕获 URL 中的值，把值放在括号里。
- 无需使用前导斜线，每个 URL 本身就有。例如，是 ^reviews，而非 ^/reviews。
- 每个正则表达式字符串前面的 'r' 是可选的，但是建议加上。它的作用是告诉 Python，那是“原始”字符串，里面的一切内容都不应该转义。

示例请求：

- 对 /reviews/2005/03/ 的请求匹配上述列表中的第三个条目。Django 调用 views.month_archive(request, '2005', '03') 函数。
- /reviews/2005/3/ 不匹配任何 URL 模式，因为列表中的第三个条目要求月份有两个数字。
- /reviews/2003/ 匹配列表中的第一个模式，而不是第二个，因为模式按顺序测试，而第一个就能让测试通过。你可以调整顺序，添加类似这样的特例。

- /reviews/2003 不匹配任何模式，因为每个模式都要求 URL 的末尾有一条斜线。
- /reviews/2003/03/03/ 匹配最后一个模式。Django 调用 views.review_detail(request, '2003', '03', '03') 函数。

下述 URL 配置与前面的作用一样，不过换用具名分组了：

```
from django.conf.urls import url
from . import views

urlpatterns = [
    url(r'^reviews/2003/$', views.special_case_2003),
    url(r'^reviews/(?P<year>[0-9]{4})/$', views.year_archive),
    url(r'^reviews/(?P<year>[0-9]{4})/(?P<month>[0-9]{2})/$',
        views.month_archive),
    url(r'^reviews/(?P<year>[0-9]{4})/(?P<month>[0-9]{2})/(?P<day>[0-9]{2})/$',
        views.review_detail),
]
```

这个 URL 配置与前一个实现的 URL 模式完全一样，只不过有一处不同：捕获的值以关键字参数传给视图函数，而不是位置参数。例如：

- 对 /reviews/2005/03/ 的请求调用 views.month_archive(request,year='2005',month='03') 函数，而不是 views.month_archive(request,'2005','03')。
- 对 /reviews/2003/03/03/ 的请求调用 views.review_detail(request,year='2003',month='03',day='03') 函数。

在实际运用中，这样做的好处是 URL 配置的意图稍微明显一些，而且不容易出现由于参数顺序不当导致的缺陷，因为视图函数定义中的参数顺序可以调整。当然，这也牺牲了一些简洁性，有些开发者觉得具名分组句法不美观，而且太啰嗦。

匹配/分组算法

URL 配置解析器解析正则表达式中具名分组和非具名分组所采用的算法如下：

1. 如果有具名分组，使用具名分组，忽略非具名分组。
2. 否则，以位置参数传递所有非具名分组。

不论如何，额外的关键字参数都会传给视图。

7.1.4 URL 配置搜索的范围

URL 配置搜索的是所请求的 URL，而且把它视作普通的 Python 字符串。搜索的范围不包括 GET 或 POST 参数，抑或域名。例如对 http://www.example.com/myapp/ 的请求，URL 配置只查找 myapp/；对 http://www.example.com/myapp/?page=3 的请求，URL 配置只查找 myapp/。URL 配置不关心请求方法。也就是说，相同 URL 的所有请求方法（POST、GET、HEAD，等等）都交由同一个视图函数处理。

7.1.5 捕获的参数始终是字符串

不管正则表达式匹配的是什么类型，捕获的每个参数都以普通的 Python 字符串传给视图。对 URL 配置中的这一行来说：

```
url(r'^reviews/(?P<year>[0-9]{4})/$', views.year_archive),
```

虽然 [0-9]{4} 只匹配字符串中的整数，但是传给 views.year_archive() 视图函数的 year 参数是字符串，而不是整数。

7.1.6 为视图的参数指定默认值

一个常用的技巧是为视图的参数指定默认值。以下面的 URL 配置为例：

```
# URL 配置
from django.conf.urls import url
from . import views

urlpatterns = [
    url(r'^reviews/$', views.page),
    url(r'^reviews/page(?P<num>[0-9]+)/$', views.page),
]

# 视图（在 reviews/views.py 文件中）
def page(request, num="1"):
    # 输出指定数量的书评
```

在上述示例中，两个 URL 模式指向同一个视图，即 views.page，但是第一个模式没有从 URL 中匹配任何内容。如果匹配第一个模式，page() 函数使用 num 的默认值，即 "1"；如果匹配第二个模式，page() 函数使用正则表达式匹配的 num 值。

7.2 性能

urlpatterns 中的每个正则表达式在首次访问时编译，因此系统的速度异常得快。

旁注 7-1：关键字参数与位置参数

调用 Python 函数时可以使用关键字参数，也可以使用位置参数，而且有些时候二者同时使用。使用关键字参数时，参数的名称和值一起传递；使用位置参数时，只传递值，而不明确指定哪个参数匹配哪个值，二者的关系由参数的顺序确定。

例如，对下面这个简单的函数来说：

```
def sell(item, price, quantity):
    print "Selling %s unit(s) of %s at %s" % (quantity, item, price)
```

如果使用位置参数，要按照函数定义中指定的顺序传入：

```
sell('Socks', '$2.50', 6)
```

如果使用关键字参数，要把参数的名称与值一起传入。下面各个调用是等价的：

```
sell(item='Socks', price='$2.50', quantity=6)
sell(item='Socks', quantity=6, price='$2.50')
sell(price='$2.50', item='Socks', quantity=6)
sell(price='$2.50', quantity=6, item='Socks')
```

```
sell(quantity=6, item='Socks', price='$2.50')
sell(quantity=6, price='$2.50', item='Socks')
```

最后，还可以混用关键字参数和位置参数，只要把位置参数放在关键字参数前面就行。下面各个调用与前面的等价：

```
sell('Socks', '$2.50', quantity=6)
sell('Socks', price='$2.50', quantity=6)
sell('Socks', quantity=6, price='$2.50')
```

7.3 错误处理

找不到匹配所请求 URL 的正则表达式或有异常抛出时，Django 会调用一个错误处理视图。具体使用的视图由四个参数指定。这四个参数是：

- handler404
- handler500
- handler403
- handler400

对多数项目来说，使用默认的处理视图应该就够了；然而，如果想定制，也可以把它们设为其他值。这四个参数的值在根 URL 配置中设定，在其他位置设定无效。设定的值必须是可调用的对象，或者是表示完整的 Python 导入路径的字符串，指向处理相应错误的视图。

7.4 引入其他 URL 配置

urlpatterns 在任何位置都可以“引入”其他 URL 配置模块。通过这一行为可以把一些 URL 放在另一些名下。例如，下面是 Django 项目的网站的 URL 配置，从其他位置引入了一些 URL 配置：

```
from django.conf.urls import include, url

urlpatterns = [
    # ...
    url(r'^community/', include('django_website.aggregator.urls')),
    url(r'^contact/', include('django_website.contact.urls')),
    # ...
]
```

注意，这里的正则表达式没有 $（匹配字符串末尾的符号），但是末尾有斜线。Django 遇到 include() 时，会把截至那一位置匹配的 URL 截断，把余下的字符串传给引入它的 URL 配置，做进一步处理。此外，还可以使用 url() 引入额外的 URL 模式。以下述 URL 配置为例：

```
from django.conf.urls import include, url
from apps.main import views as main_views
from credit import views as credit_views

extra_patterns = [
    url(r'^reports/(?P<id>[0-9]+)/$', credit_views.report),
    url(r'^charge/$', credit_views.charge),
```

```
]

urlpatterns = [
    url(r'^$', main_views.homepage),
    url(r'^help/', include('apps.help.urls')),
    url(r'^credit/', include(extra_patterns)),
]
```

这里，/credit/reports/ URL 由 credit.views.report() 视图处理。利用这一行为可以去除 URL 配置中的重复，在多处使用相同的模式前缀。来看这个 URL 配置：

```
from django.conf.urls import url
from . import views

urlpatterns = [
    url(r'^(?P<page_slug>\w+)-(?P<page_id>\w+)/history/$',
        views.history),
    url(r'^(?P<page_slug>\w+)-(?P<page_id>\w+)/edit/$',
views.edit),
    url(r'^(?P<page_slug>\w+)-(?P<page_id>\w+)/discuss/$',
        views.discuss),
    url(r'^(?P<page_slug>\w+)-(?P<page_id>\w+)/permissions/$',
        views.permissions),
]
```

我们可以改进这个 URL 配置，把共用的路径前缀提取出来，再把不同的部分放在其后：

```
from django.conf.urls import include, url
from . import views

urlpatterns = [
    url(r'^(?P<page_slug>\w+)-(?P<page_id>\w+)/',
        include([
            url(r'^history/$', views.history),
            url(r'^edit/$', views.edit),
            url(r'^discuss/$', views.discuss),
            url(r'^permissions/$', views.permissions),
        ])
    ),
]
```

7.4.1 捕获的参数

被引入的 URL 配置会接收到父级 URL 配置捕获的参数，因此下述示例是有效的：

```
# settings/urls/main.py
from django.conf.urls import include, url

urlpatterns = [
    url(r'^(?P<username>\w+)/reviews/', include('foo.urls.reviews')),
]

# foo/urls/reviews.py
```

```
from django.conf.urls import url
from . import views

urlpatterns = [
    url(r'^$', views.reviews.index),
    url(r'^archive/$', views.reviews.archive),
]
```

这里，捕获的 "username" 变量能顺利传给被引入的 URL 配置。

7.5 给视图函数传递额外参数

URL 配置允许向视图函数传递额外的参数，这些参数放在一个 Python 字典中。django.conf.urls.url() 函数的第三个参数是可选的，如果指定，应该是一个字典，指定要传给视图函数的额外关键字参数及其值。例如：

```
from django.conf.urls import url
from . import views

urlpatterns = [
    url(r'^reviews/(?P<year>[0-9]{4})/$',
        views.year_archive,
        {'foo': 'bar'}
    ),
]
```

对这个示例来说，请求 /reviews/2005/ 时，Django 调用 views.year_archive(request, year='2005', foo='bar')。聚合（syndication）框架通过这种方式把元数据和选项传给视图（参见第 14 章）。

处理冲突

有可能 URL 模式捕获了具名关键字参数，又在第三个参数中传递同名的参数。此时，Django 使用字典中的参数，而不是从 URL 中捕获的参数。

7.5.1 给 include() 传递额外参数

同样，也可以为 include() 传递额外参数。此时，被引入的 URL 配置中的每一行都将收到额外的参数。例如，下述两个 URL 配置的作用是一样的。

第一个 URL 配置：

```
# main.py
from django.conf.urls import include, url

urlpatterns = [
    url(r'^reviews/', include('inner'), {'reviewid': 3}),
]

# inner.py
from django.conf.urls import url
from mysite import views
```

```python
urlpatterns = [
    url(r'^archive/$', views.archive),
    url(r'^about/$', views.about),
]
```

第二个 URL 配置：

```python
# main.py
from django.conf.urls import include, url
from mysite import views

urlpatterns = [
    url(r'^reviews/', include('inner')),
]

# inner.py
from django.conf.urls import url

urlpatterns = [
    url(r'^archive/$', views.archive, {'reviewid': 3}),
    url(r'^about/$', views.about, {'reviewid': 3}),
]
```

注意，不管 URL 模式是否能接收参数，额外的参数都将传给被引入的 URL 配置的每个模式。鉴于此，仅当确定被引入的 URL 配置中的每个视图都接收传入的额外参数时，才应该这么做。

7.6 反向解析 URL

在 Django 项目中经常需要获取 URL 的最终形式，这么做是为了在生成的内容中嵌入 URL（视图和静态资源的 URL、呈现给用户的 URL，等等），或者在服务器端处理导航流程（重定向等）。

此时，一定不能硬编码 URL（费时、不可伸缩，而且容易出错），或者参照 URL 配置创造一种生成 URL 的机制，因为这样容易导致线上 URL 失效。也就是说，我们需要一种不自我重复的机制。

这种机制的一个优点是，改进 URL 设计之后无需在项目的源码中大范围搜索，替换掉过期的 URL。目前，我们能获取的信息有负责处理 URL 的视图标识（即视图名称），以及查找正确 URL 所需的视图参数类型（位置参数或关键字参数）和值。

Django 提供了一种方案，只需在 URL 映射中设计 URL。我们为其提供 URL 配置，然后可以双向使用：

- 从用户（浏览器）请求的 URL 开始，这个方案能调用正确的 Django 视图，并从 URL 中提取可能需要的参数及其值，传给视图。
- 从 Django 视图对应的标识以及可能传入的参数值开始，获取相应的 URL。

第一点就是我们目前所讨论的处理方式。第二点称为反向解析 URL、反向匹配 URL、反向查找 URL 或 URL 反转。

Django 在不同的层中提供了执行 URL 反转所需的工具：

- 在模板中，使用 `url` 模板标签。
- 在 Python 代码中，使用 `django.core.urlresolvers.reverse()` 函数。

- 在处理 Django 模型实例 URL 相关的高层代码中，使用 `get_absolute_url()` 方法。

7.6.1 示例

仍以下述 URL 配置为例：

```
from django.conf.urls import url
from . import views

urlpatterns = [
    #...
    url(r'^reviews/([0-9]{4})/$', views.year_archive,
        name='reviews-year-archive'),
    #...
]
```

根据这个设计，*nnnn* 年的存档对应的 URL 是 `/reviews/nnnn/`。在模板中可以使用下述代码获取这个 URL：

```
<a href="{% url 'reviews-year-archive' 2012 %}">2012 Archive</a>

{# 或者把年份存储在一个模板上下文变量中：#}

<ul>
{% for yearvar in year_list %}
<li><a href="{% url 'reviews-year-archive' yearvar %}">{{
yearvar }} Archive</a></li>
{% endfor %}
</ul>
```

在 Python 代码中则要这么做：

```
from django.core.urlresolvers import reverse
from django.http import HttpResponseRedirect

def redirect_to_year(request):
    # ...
    year = 2012
    # ...
    return HttpResponseRedirect(reverse('reviews-year-archive', args=(year,)))
```

如果基于某些原因，想修改显示年份书评存档的 URL，只需修改 URL 配置中的条目。某些情况下，视图是通用的，URL 和视图之间存在多对一关系，URL 反转时视图名称不足以唯一标识。Django 为此提供的解决方案参阅下一节。

7.7 为 URL 模式命名

为了执行 URL 反转，要像前述示例那样为 URL 模式命名。URL 模式的名称可以包含任何字符串，而不限定于必须是有效的 Python 标识符。为 URL 模式命名时，要确保不与其他应用中的名称冲突。如果你把 URL 模式命名为 `comment`，而另一个应用也这么做，在模板中使用这个名称时就无法确定该生成哪个 URL。为了减少冲突，可以为 URL 模式的名称加上前缀，比如说使用应用程序的名称。我们建议使用 `myapp-comment`，而不是 `comment`。

7.8 URL 命名空间

URL 命名空间在反转具名 URL 模式时具有唯一确定性，即便不同的应用使用相同的名称也不怕。鉴于此，第三方应用最好始终把 URL 放在命名空间中。类似地，URL 命名空间在部署多个应用程序实例时也能正确反转 URL。也就是说，同一个应用程序的多个实例使用相同的 URL 模式名称，而通过命名空间可以把它们区分开。

正确使用 URL 命名空间的 Django 应用程序可以在同一个网站中多次部署。例如，`django.contrib.admin` 中有个 `AdminSite` 类，可以轻易部署多个管理后台。URL 命名空间分为两部分，而且都是字符串：

1. 应用命名空间。指明应用的名称。一个应用的每个实例都具有相同的应用命名空间。例如，你可能猜到了，Django 管理后台的应用命名空间是 `admin`。
2. 实例命名空间。标识具体的应用程序实例。实例命名空间在整个项目范围内应该是唯一的。不过，实例命名空间可以与应用命名空间相同，供应用的默认实例使用。例如，Django 管理后台实例的默认实例命名空间是 `admin`。

命名空间中的 URL 使用 : 运算符指定。例如，管理后台的主页使用 `admin:index` 引用。其中，`admin` 是命名空间，`index` 是 URL 的名称。

命名空间还可以嵌套。`members:reviews:index` 在命名空间 `members` 中查找命名空间 `reviews`，再在里面查找 `index` URL。

7.8.1 反转命名空间中的 URL

解析命名空间中的 URL 时（如 `reviews:index`），Django 先分解完全限定的名称，然后尝试做下述查找：

1. 首先，Django 查找有没有匹配的应用命名空间（这里的 `reviews`）。为此，会产出那个应用的实例列表。
2. 如果有这么一个应用实例，Django 返回它的 URL 解析程序。当前应用可以通过请求的一个属性指定。预期有多个部署实例的应用应该在处理的请求上设定 `current_app` 属性。
3. 当前应用也可以手动指定，方法是作为参数传给 `reverse()` 函数。
4. 如果没有当前应用，Django 查找默认的应用实例。默认应用实例是指实例命名空间与应用命名空间匹配的实例（在这里是指名为 `reviews` 的 `reviews` 实例）。
5. 如果没有默认的应用实例，Django 选中最后部署的应用实例，而不管实例的名称。
6. 如果第 1 步找不到匹配的应用命名空间，Django 直接把它视作实例命名空间查找。

对嵌套的命名空间来说，上述步骤不断重复，直到视图名称为止。视图名称在找到的命名空间中解析成 URL。

7.8.2 URL 命名空间和引入的 URL 配置

把引入的 URL 配置放入命名空间中有两种方式。第一种，在 URL 模式中为 `include()` 提供应用和实例命名空间。例如：

```
url(r'^reviews/', include('reviews.urls',
                          namespace='author-reviews',
                          app_name='reviews')),
```

上述代码把 `reviews.urls` 中定义的 URL 放在应用命名空间 `reviews` 中，放在实例命名空间 `author-reviews`

中。第二种方式是，引入包含命名空间数据的对象。如果使用 include() 引入一组 url() 实例，那个对象中的 URL 都添加到全局命名空间中。然而，include() 的参数还可以是一个三元素元组，其内容如下：

```
(<list of url() instances>, <application namespace>, <instance namespace>)
```

例如：

```
from django.conf.urls import include, url
from . import views

reviews_patterns = [
    url(r'^$', views.IndexView.as_view(), name='index'),
    url(r'^(?P<pk>\d+)/$', views.DetailView.as_view(), name='detail'),
]

url(r'^reviews/', include((reviews_patterns, 'reviews', 'author-reviews'))),
```

上述代码把指定的 URL 模式引入指定的应用和实例命名空间中。例如，部署的 Django 管理后台是 AdminSite 的实例。AdminSite 对象有个 urls 属性，它的值是一个三元素元组，包含相应管理后台的所有 URL 模式，以及应用命名空间 'admin' 和管理后台实例的名称。部署管理后台实例时，引入项目的 urlpatterns 中的就是这个 urls 属性。

记得要把一个元组传给 include()。如果直接传入三个参数，例如 include(reviews_patterns, 'reviews', 'author-reviews')，Django 不会抛出错误，但是根据 include() 的签名，'reviews' 是实例命名空间，'author-reviews' 是应用命名空间，而正确的顺序应该反过来。

7.9 接下来

本章介绍了很多高级的视图和 URL 配置技巧。接下来，在第 8 章，我们将讨论 Django 模板系统的高级话题。

第 8 章 高级模板技术

虽然你基本上是作为编写者去使用 Django 的模板语言，但是有时可能想定制或扩展模板引擎，例如实现没有自带的功能，或者以某种方式简化工作。

本章深入讨论 Django 的模板系统，让你知道如何扩展这一系统，也满足你对内部运作机制的好奇心。此外，本章还涵盖自动转义特性。这是一项安全措施，在使用 Django 的过程中你肯定会注意到它的存在。

8.1 模板语言回顾

首先，我们快速回顾一下第 3 章介绍的几个术语：

- 模板是文本文档或普通的 Python 字符串，使用 Django 模板语言标记。模板中有模板标签和变量。
- 模板标签是模板中的一种符号，用于做特定的事情。这样定义相当晦涩。我们来举些例子：模板标签可以生成内容、用做控制结构（if 语句或 for 循环）、从数据库中获取内容，或者访问其他模板标签。模板标签放在 {% 和 %} 之间：

```
{% if is_logged_in %}
    Thanks for logging in!
{% else %}
    Please log in.
{% endif %}
```

- 变量也是模板中的一种符号，用于输出值。变量放在 {{ 和 }} 之间：

```
My first name is {{ first_name }}. My last name is {{ last_name }}.
```

- 上下文是传给模板的名值映射（类似于 Python 字典）。
- 模板渲染上下文的过程是把变量所在的位置替换成上下文中的值，并执行所有模板标签。

关于这些术语的更多说明，参阅第 3 章。本章余下的内容讨论扩展模板引擎的方式。不过，在此之前我们要简单说明一下第 3 章没有涵盖的内部细节。

8.2 RequestContext 和上下文处理器

模板要在上下文中渲染。上下文是 `django.template.Context` 的实例，不过 Django 还提供了一个子类，`django.template.RequestContext`，其行为稍有不同。

`RequestContext` 默认为模板上下文添加很多变量，例如 `HttpRequest` 对象或当前登录用户的信息。

使用 `render()` 快捷方式时，如果没有明确传入其他上下文，默认使用 `RequestContext`。来看下面两个视图：

```
from django.template import loader, Context

def view_1(request):
    # ...
```

```
        t = loader.get_template('template1.html')
        c = Context({
            'app': 'My app',
            'user': request.user,
            'ip_address': request.META['REMOTE_ADDR'],
            'message': 'I am view 1.'
        })
        return t.render(c)

    def view_2(request):
        # ...
        t = loader.get_template('template2.html')
        c = Context({
            'app': 'My app',
            'user': request.user,
            'ip_address': request.META['REMOTE_ADDR'],
            'message': 'I am the second view.'
        })
        return t.render(c)
```

（注意，这里我故意没有使用 `render()` 的快捷方式，而是自己动手加载模板、构建上下文对象，再渲染模板。我把这些步骤写出来，是为了便于讨论。）

在这两个视图中，我们向模板传入了相同的三个变量：`app`、`user` 和 `ip_address`。如果能去掉这些重复不是更好吗？`RequestContext` 和上下文处理器就是为解决这种问题而出现的。上下文处理器用于指定自动在各个上下文中设定的变量，这样就无需每次调用 `render()` 时都指定。

为此，渲染模板时要把 `Context` 换成 `RequestContext`。使用上下文处理器最低层的做法是创建一些处理器，将其传给 `RequestContext`。使用上下文处理器改写上述示例得到的代码如下：

```
    from django.template import loader, RequestContext

    def custom_proc(request):
        # 一个上下文处理器，提供 'app'、'user' 和 'ip_address'
        return {
            'app': 'My app',
            'user': request.user,
            'ip_address': request.META['REMOTE_ADDR']
        }

    def view_1(request):
        # ...
        t = loader.get_template('template1.html')
        c = RequestContext(request,
                           {'message': 'I am view 1.'},
                           processors=[custom_proc])
        return t.render(c)

    def view_2(request):
        # ...
        t = loader.get_template('template2.html')
        c = RequestContext(request,
                           {'message': 'I am the second view.'},
```

```
                           processors=[custom_proc])
    return t.render(c)
```

下面分析一下这段代码：

- 首先，定义函数 custom_proc。这是一个上下文处理器，它的参数是一个 HttpRequest 对象，返回值是一个字典，包含要在模板上下文中使用的变量。就这么简单。
- 我们修改了那两个视图函数，把 Context 换成 RequestContext。构建这种上下文的方式有两处不同：其一，RequestContext 要求第一个参数必须是一个 HttpRequest 对象，即传给视图函数的那个参数（request）；其二，RequestContext 接受可选的 processors 参数，其值是要使用的上下文处理器列表或元组。这里，我们传入的是前面定义的那个处理器 custom_proc。
- 现在，各个视图在构建上下文时无需包含 app、user 或 ip_address，因为 custom_proc 已经提供。
- 如果需要，视图仍可以提供其他模板变量。这里，我们为两个视图设定了不同的 message 模板变量。

我在第 3 章介绍了 render() 快捷方式，这样能避免在模板中自己动手调用 loader.get_template()、创建上下文，再调用 render() 方法。

为了说明上下文处理器的低层工作方式，上述示例没有使用 render() 快捷方式。但是，使用上下文处理器时可以这么做，也推荐这么做。为此，要使用 context_instance 参数，如下所示：

```
from django.shortcuts import render
from django.template import RequestContext

def custom_proc(request):
    # 一个上下文处理器，提供 'app'、'user' 和 'ip_address'
    return {
        'app': 'My app',
        'user': request.user,
        'ip_address': request.META['REMOTE_ADDR']
    }

def view_1(request):
    # ...
    return render(request, 'template1.html',
                  {'message': 'I am view 1.'},
                  context_instance=RequestContext(
                      request, processors=[custom_proc]
                  )
    )

def view_2(request):
    # ...
    return render(request, 'template2.html',
                  {'message': 'I am the second view.'},
                  context_instance=RequestContext(
                      request, processors=[custom_proc]
                  )
    )
```

在这两个视图中，我们把渲染模板的代码缩减到只有一行（有换行）。这是进步，但是考虑到代码的简洁性，这么做无异于走向另一个极端。我们去掉了数据（模板变量）中的重复，但是增加了代码（调用 processors 的代码）中的重复。

如果每一次都要输入 processors，使用上下文处理器省不了多少事。鉴于此，Django 提供了全局上下文处理器。context_processors 设置（在 settings.py 文件中）指明始终提供给 RequestContext 的上下文处理器。这样便不用每次使用 RequestContext 时都指定 processors 参数了。

context_processors 的默认值如下：

```
'context_processors': [
    'django.template.context_processors.debug',
    'django.template.context_processors.request',
    'django.contrib.auth.context_processors.auth',
    'django.contrib.messages.context_processors.messages',
],
```

这是一个可调用对象列表，接口与前文定义的 custom_proc 函数一样——参数是一个请求对象，返回值是一个字典，包含要合并到上下文中的变量。注意，context_processors 中的值是字符串，因此处理器要在 Python 路径中（这样才能在设置中引用）。

指定的各个处理器按顺序运用。因此，如果一个处理器向上下文中添加了一个变量，而后面一个处理器又添加了一个同名变量，那么后一个将覆盖前一个。Django 提供了几个简单的上下文处理器，包含默认启用的那几个。下面几小节详述。

8.2.1 auth

django.contrib.auth.context_processors.auth

启用这个处理器后，RequestContext 中将包含下述变量：

- user：auth.User 的实例，表示当前登录的用户（如未登录，是 AnonymousUser 实例）。
- perms：django.contrib.auth.context_processors.PermWrapper 实例，表示当前登录用户拥有的权限。

8.2.2 debug

django.template.context_processors.debug

启用这个处理器后，RequestContext 中将包含下面两个变量，但前提是 DEBUG 设置的值是 True，而且 INTERNAL_IPS 设置中包含请求的 IP 地址（request.META['REMOTE_ADDR']）：

- debug：True。可以在模板中测试是否在 DEBUG 模式中。
- sql_queries：{'sql': …, 'time': …} 字典构成的列表，表示处理请求的过程中执行的 SQL 查询及其用时。列表中的值按查询的执行顺序排列，在访问时惰性生成。

8.2.3 i18n

django.template.context_processors.i18n

启用这个处理器后，RequestContext 中将包含下面两个变量：

- LANGUAGES：LANGUAGES 设置的值。
- LANGUAGE_CODE：如果 request.LANGUAGE_CODE 存在，返回它的值；否则返回 LANGUAGE_CODE 设置的值。

8.2.4 media

django.template.context_processors.media

启用这个处理器后，RequestContext 中将包含 MEDIA_URL 变量，提供 MEDIA_URL 设置的值。

8.2.5 static

django.template.context_processors.static

启用这个处理程序后，RequestContext 中将包含 STATIC_URL 变量，提供 STATIC_URL 设置的值。

8.2.6 csrf

django.template.context_processors.csrf

这个处理器添加一个令牌，供 csrf_token 模板标签使用，用于防范跨站请求伪造（参见第 19 章）。

8.2.7 request

django.template.context_processors.request

启用这个处理器后，RequestContext 中将包含 request 变量，它的值是当前的 HttpRequest 对象。

8.2.8 messages

django.contrib.messages.context_processors.messages

启用这个处理器后，RequestContext 中将包含下面两个变量：

- messages：消息框架设定的消息列表（里面的值是字符串）。
- DEFAULT_MESSAGE_LEVELS：消息等级名称到数字值的映射。

8.3 自定义上下文处理器的指导方针

上下文处理器的接口十分简单，它就是普通的 Python 函数，有一个参数，是一个 HttpRequest 对象，返回一个字典，用于添加到模板上下文中。上下文处理器必须返回一个字典。下面是自定义处理器的几个小贴士：

- 上下文处理器应该尽量负责较少的功能。处理器易于合用，因此应该把功能分成逻辑片段，供以后复用。
- 记住，TEMPLATE_CONTEXT_PROCESSORS 中列出的各个上下文处理器在那个设置文件管辖的每个模板中都可用，因此应该为变量挑选不易与模板自身设定的变量冲突的名称。因为变量区分大小写，所以让处理器提供的变量都使用大写不失为一个好主意。
- 自定义的上下文处理器可以放在代码基的任何位置。Django 只关心 TEMPLATES 设置中的 'context_processors' 选项或者 Engine 的 context_processors 参数（直接使用 Engine 时）有没有指向你的上下文处理器。尽管如此，约定的做法是把上下文处理器保存在应用或项目中一个名为 context_processors.py 的文件中。

8.4 自动转义 HTML

使用模板生成 HTML 时，变量的值可能包含特殊的字符，对得到的 HTML 产生影响。对下述模板片段来说：

```
Hello, {{ name }}.
```

乍一看，这样显示用户的名字没什么危害。但是，如果用户输入的名字是这样的呢：

```
<script>alert('hello')</script>
```

此时，渲染模板后得到的结果是：

```
Hello, <script>alert('hello')</script>
```

因此，浏览器会弹出一个对话框。同样，如果名字中包含 `'<'` 符号呢：

```
<b>username
```

此时，渲染模板后得到的结果是：

```
Hello, <b>username
```

这样，网页中的后续内容都会显示为粗体。显然，用户提交的数据不该盲目信任，不能直接插入网页，因为恶意用户可以利用这种漏洞做些坏事。

这种安全漏洞称为*跨站脚本攻击*（Cross Site Scripting，XSS）。（安全方面的话题参见第 19 章。）为了避免这种漏洞，有两个选择：

1. 每个不信任的变量都传给 `escape` 过滤器，把有潜在危害的 HTML 字符转换成无危害的。Django 起初的几年默认采用这种方案，它的问题是把责任强加到开发者或模板编写者身上了，你要确保转义一切。但是，忘记转义数据是常事。
2. 利用 Django 的自动转义 HTML 特性。本节余下的内容说明自动转义的工作方式。

Django 默认转义模板中的每个变量标签。具体而言，转义的是下面五个字符：

- `<` 转换成 `<`
- `>` 转换成 `>`
- `'`（单引号）转换成 `'`
- `"`（双引号）转换成 `"`
- `&` 转换成 `&`

我们要再次强调，这个行为默认已启用。如果使用 Django 的模板系统，就能受到这一措施的保护。

8.4.1 如何禁用

自动转义可以在整站禁用、在模板层禁用或在变量层禁用。为什么要禁用呢？因为有时候想把模板变量包含的数据渲染成原始 HTML，无需转义。

例如，可能在数据库中存储了授信的 HTML blob，想直接将其插入模板。或者，使用 Django 的模板系统生成非 HTML 文本，例如电子邮件。

在单个变量中禁用

若想在单个变量中禁用自动转义，使用 safe 过滤器：

```
This will be escaped: {{ data }}
This will not be escaped: {{ data|safe }}
```

这个过滤器的作用可以理解为“无需转义，可以放心使用”，或者“可以安全解释为 HTML”。这里，如果 data 中包含 '<b> '，得到的结果为：

```
This will be escaped: &lt;b&gt;
This will not be escaped: <b>
```

在模板中的块里禁用

若想在模板中控制自动转义行为，把模板（或其中一部分）放在 autoescape 标签里，如下所示：

```
{% autoescape off %}
    Hello {{ name }}
{% endautoescape %}
```

autoescape 标签的参数为 on 或 off。有时需要强制自动转义，有时则想禁用。下面举个例子：

```
Auto-escaping is on by default. Hello {{ name }}

{% autoescape off %}
    This will not be auto-escaped: {{ data }}.

    Nor this: {{ other_data }}
    {% autoescape on %}
        Auto-escaping applies again: {{ name }}
    {% endautoescape %}
{% endautoescape %}
```

autoescape 标签的作用能延伸到扩展当前模板的模板，以及通过 include 标签引入的模板——这一点与其他块标签一样。例如：

```
# base.html

{% autoescape off %}
    <h1>{% block title %}{% endblock %}</h1>
    {% block content %}
    {% endblock %}
{% endautoescape %}

# child.html

{% extends "base.html" %}
{% block title %}This & that{% endblock %}
{% block content %}{{ greeting }}{% endblock %}
```

基模板禁用了自动转义，因此子模板也将禁用。greeting 变量包含字符串 Hello! 时，渲染得到的 HTML 如下：

```
<h1>This & that</h1>
```

```
<b>Hello!</b>
```

一般来说，模板编写人无需过多关注自动转义。Python 侧的开发者（编写视图和自定义过滤器的人）要考虑哪些情况下不用转义数据，并且为数据做好相应的标记，这样渲染模板得到的结果才符合预期。

如果使用模板的环境不确定是否启用了自动转义，应该添加 `escape` 过滤器，用于转义需要转义的变量。如果启用了自动转义，`escape` 过滤器会再次转义数据，但是这样做并无坏处，因为 `escape` 过滤器对自动转义过的变量没有影响。

8.4.2 自动转义过滤器参数中的字符串字面量

前面说过，过滤器的参数可以是字符串，例如：

```
{{ data|default:"This is a string literal." }}
```

字符串字面量插入模板时不会自动转义，就像是经过 `safe` 过滤器处理过了一样。这样处理的原因是，模板编写人员负责控制字符串字面量中的内容，他们在编写模板时可以确保正确转义了文本。

因此，你应该编写：

```
{{ data|default:"3 &lt; 2" }}
```

而不是：

```
{{ data|default:"3 < 2" }} {# <-- 不对！别这么做。#}
```

从变量中得出的数据不是如此。变量的内容在需要时会自动转义，因为它们不在模板编写人的控制范围内。

8.5 模板加载内部机制

通常，我们把模板保存在文件系统中的文件里，而不直接使用低层的 `Template` API。建议把模板保存在一个专门的目录中。Django 在多个位置搜索模板目录，具体是哪些取决于模板加载设置（参见 8.5.2 节），但是指定模板目录最基本的方式是使用 `DIRS` 选项。

8.5.1 DIRS 选项

告诉 Django 模板目录有哪些的方法是使用设置文件中 `TEMPLATES` 设置的 `DIRS` 选项，或者是 `Engine` 的 `dirs` 参数。这个选项的值是一个字符串列表，包含指向模板目录的完整路径：

```
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [
            '/home/html/templates/lawrence.com',
            '/home/html/templates/default',
        ],
    },
]
```

模板可以放在任何位置，只要 Web 服务器有权限读取目录及里面的模板即可。模板的扩展名不限，可以是 `.html` 或 `.txt`，甚至可以没有。注意，这里的路径应该使用 Unix 风格的正斜线，即便在 Windows 中也是如此。

8.5.2 加载器的类型

Django 默认使用基于文件系统的模板加载器（loader），不过除此之外 Django 还自带了几个其他的加载器，它们知道如何从其他源加载模板。其中，最常使用的是应用目录加载器，参见下面的说明。

文件系统加载器

`filesystem.Loader`

根据 `DIRS` 的值，从文件系统中加载模板。这是默认启用的加载器。然而，如果不设定 `DIRS` 选项，这个加载器找不到任何模板。

```
TEMPLATES = [{
    'BACKEND': 'django.template.backends.django.DjangoTemplates',
    'DIRS': [os.path.join(BASE_DIR, 'templates')],
}]
```

应用目录加载器

`app_directories.Loader`

从文件系统中的 Django 应用里加载模板。这个加载器在 `INSTALLED_APPS` 列出的各个应用中查找 `templates` 子目录。如果找到，Django 在其中查找模板。这意味着，应用可以自带模板。通过这一行为，便于分发带默认模板的 Django 应用。例如，对下面的设置来说：

```
INSTALLED_APPS = ['myproject.reviews', 'myproject.music']
```

`get_template('foo.html')` 会按顺序在下述目录中查找 `foo.html`：

- `/path/to/myproject/reviews/templates/`
- `/path/to/myproject/music/templates/`

而且使用最先找到的那个。

鉴于此，`INSTALLED_APPS` 中罗列应用的顺序是十分重要的！

假如你想自定义 Django 的管理后台，你可以选择覆盖 `django.contrib.admin` 中的 `admin/base_site.html` 模板，把自定义的模板 `admin/base_site.html` 保存在 `myproject.reviews` 应用中。

此时，在 `INSTALLED_APPS` 中必须把 `myproject.reviews` 放在 `django.contrib.admin` 前面，否则将先加载 `django.contrib.admin`，你自定义的模板就被忽略了。

注意，加载器首次运行时会执行一项优化措施：缓存 `INSTALLED_APPS` 中有 `templates` 子目录的包。

只需把 `APP_DIRS` 选项设为 `True` 即可启用这个加载器：

```
TEMPLATES = [{
    'BACKEND': 'django.template.backends.django.DjangoTemplates',
    'APP_DIRS': True,
}]
```

其他加载器

此外还有几个模板加载器：

- django.template.loaders.eggs.Loader
- django.template.loaders.cached.Loader
- django.template.loaders.locmem.Loader

这些加载器默认禁用，不过可以在 TEMPLATES 设置中为 DjangoTemplates 后端添加 'loaders' 选项，或者把 loaders 参数传给 Engine 启用。这些高级加载器的详细说明，以及构建自定义加载器的方法参见 Django 项目的网站。

8.6 扩展模板系统

现在，我们了解了一些模板系统的内部细节，下面来看如何扩展这个系统。对模板系统的定制，基本上是自定义模板标签和（或）过滤器。虽然 Django 模板语言自带了很多内建的标签和过滤器，但是有时也需要自己创建一些标签和过滤器，满足自己的需求。幸好，这并不难。

8.6.1 代码布局

自定义的模板标签和过滤器必须放在一个 Django 应用中。如果与现有应用有关，可以放在现有应用中；否则，应该专门创建一个应用存放。应用中应该有个 templatetags 目录，与 models.py、views.py 等文件放在同一级。如果没有这个目录，创建一个，别忘了 __init__.py 文件，这样才能保证所在目录是一个 Python 包。

添加这个模块之后，要重启服务器方能在模板中使用自定义的标签或过滤器。自定义的标签和过滤器在 templatetags 目录里的一个模块中。

模块文件的名称是加载标签所用的名称，所以要小心选择，别与其他应用中的自定义标签和过滤器冲突了。

假如自定义的标签（过滤器）放在 review_extras.py 文件中，应用的布局可能是下面这样：

```
reviews/
    __init__.py
    models.py
    templatetags/
        __init__.py
        review_extras.py
    views.py
```

在模板中则这样使用：

```
{% load review_extras %}
```

包含自定义标签的应用必须在 INSTALLED_APPS 中列出，这样 {% load %} 标签才能起作用。

> **背后的运作方式**
>
> 举再多的例子也不如阅读源码，学习 Django 是如何定义默认的过滤器和标签的。过滤器和标签分别在 django/template/defaultfilters.py 和 django/template/defaulttags.py 文件中。load 标签的更多信息参阅文档。

8.6.2 创建模板库

不管是自定义标签还是过滤器，第一件事都是创建模板库——这是让 Django 勾住的基本要求。

创建模板库分为两步：

- 首先，决定把模板库放在哪个 Django 应用中。如果应用是使用 `manage.py startapp` 创建的，可以把模板库放在那里；如若不然，可以专门创建一个应用，用于存放模板库。我们推荐后者，因为自定义的标签和过滤器可能对以后的项目有用。不管怎么做，一定要把应用添加到 `INSTALLED_APPS` 设置中。具体怎么做，稍后说明。
- 其次，在 Django 应用中合适的包里创建 `templatetags` 目录。这个目录应该与 `models.py`、`views.py` 等文件放在同一级。例如：

```
books/
    __init__.py
    models.py
    templatetags/
    views.py
```

在 `templatetags` 目录中创建两个空文件：`__init__.py`（告诉 Python 这是包含 Python 代码的包）和存放自定义标签（过滤器）的文件。后者的名称是加载标签所用的名称。例如，自定义的标签（过滤器）保存在 `review_extras.py` 文件中，那么在模板中要这么加载：

```
{% load review_extras %}
```

`{% load %}` 标签在 `INSTALLED_APPS` 设置中查找包含自定义标签（过滤器）的应用，而且只从应用中加载模板库。这是一个安全特性，以便在同一台电脑中存贮多个模板库的 Python 代码，而不让每个 Django 实例都能访问。

如果模板库不与任何模型或视图绑定，Django 应用中可以只有 `templatetags` 包。事实上，这也相当常见。

`templatetags` 包中的模块数量不限。记住，`{% load %}` 语句加载的是指定的 Python 模块，而不是应用。

创建存放标签（过滤器）的 Python 模块之后，剩下的就是编写 Python 代码了。代码怎么写，取决于定义的是过滤器还是标签。一个有效的标签库必须有一个名为 `register` 的模块层变量，其值是 `template.Library` 的实例。

标签和过滤器都通过这种方式注册。因此，在模块顶部要插入下述代码：

```
from django import template

register = template.Library()
```

8.7 自定义模板标签和过滤器

Django 的模板语言自带了丰富的标签和过滤器，能满足常见的表现逻辑需求。但是，这些内建的标签和过滤器可能缺少你需要的功能。

我们可以扩展模板引擎，使用 Python 自定义标签和过滤器，然后使用 `{% load %}` 标签加载，让自定义的标签和过滤器可在模板中使用。

8.7.1 自定义模板过滤器

自定义的过滤器其实就是普通的 Python 函数，接受一个或多个参数：

1. 变量的值（输入），不一定是字符串。
2. 参数的值，可以有默认值，也可以留空。

例如，对 {{ var|foo:"bar" }} 来说，传给 foo 过滤器的变量是 var，参数是 "bar"。因为模板语言没有提供异常处理功能，所以模板过滤器抛出的异常会以服务器错误体现出来。

鉴于此，模板过滤器应该避免抛出异常，而是回落到其他合理的值。如果输入明显有问题，最好还是抛出异常，以免深埋缺陷。下面是一个示例过滤器的定义：

```
def cut(value, arg):
    """Removes all values of arg from the given string"""
    return value.replace(arg, '')
```

这个过滤器的用法举例如下：

```
{{ somevariable|cut:"0" }}
```

多数过滤器没有参数。此时，在函数中留空参数即可。例如：

```
def lower(value): # 只有一个参数
    """Converts a string into all lowercase"""
    return value.lower()
```

注册自定义的过滤器

定义好过滤器之后，要使用 Library 实例注册，让 Django 的模板语言知道它的存在：

```
register.filter('cut', cut)
register.filter('lower', lower)
```

Library.filter() 有两个参数：

1. 过滤器的名称，一个字符串。
2. 负责处理过滤器的函数，一个 Python 函数（不是函数名称的字符串形式）。

register.filter() 也可以作为装饰器使用：

```
@register.filter(name='cut')
def cut(value, arg):
    return value.replace(arg, '')

@register.filter
def lower(value):
    return value.lower()
```

如果不指定 name 参数，如上面的第二个示例，Django 使用函数的名称作为过滤器的名称。最后，register.filter() 还接受三个关键字参数：is_safe、needs_autoescape 和 expects_localtime。这些参数在“过滤器和自动转义”和“过滤器和时区”两节说明。

期待字符串的模板过滤器

如果模板过滤器期望第一个参数是字符串，应该使用 stringfilter 装饰器。这样，对象在传给过滤器之前会先转换成字符串值。

```
from django import template
from django.template.defaultfilters import stringfilter

register = template.Library()

@register.filter
@stringfilter
def lower(value):
    return value.lower()
```

这里，可以把整数传给过滤器，不会抛出 AttributeError（因为整数没有 lower() 方法）。

过滤器和自动转义

自定义过滤器时，应该想想 Django 的自动转义行为对过滤器有什么影响。注意，模板代码中存在三种字符串：

- *原始字符串*是原生的 Python str 或 unicode 类型。输出时，如果启用了自动转义就转义，否则原封不动呈现出来。
- *安全字符串*是标记为安全的字符串，输出时不会转义，因为已经做了必要的转义。在客户端需要原封不动输出原始 HTML 时经常使用这种字符串。

  在内部，这种字符串是 SafeBytes 或 SafeText 类型。二者的共同基类是 SafeData，因此可以使用类似下面的代码测试：

  ```
  if isinstance(value, SafeData):

      # 对“安全的”字符串做些处理

  ...
  ```
- *标记为“需要转义”的字符串*是在输出时始终应该转义的字符串，不管在不在 autoescape 块中都是如此。然而，即使启用了自动转义，这种字符串也只转义一次。在内部，这种字符串是 EscapeBytes 或 EscapeText 类型。一般情况下，无需关注这种类型；它们只在 escape 过滤器的实现中存在。

模板过滤器分属两种情况：

1. 不在尚未呈现的结果中引入对 HTML 不安全的字符（<、>、'、" 或 &）。
2. 过滤器代码自行负责做必要的转义。在结果中引入新的 HTML 标记时必须这么做。

对第一种情况来说，可以交给 Django 的自动转义行为处理。你只需在注册过滤器函数时把 is_safe 旗标设为 True，如下所示：

```
@register.filter(is_safe=True)
def myfilter(value):
    return value
```

这个旗标告诉 Django，如果传入过滤器的是“安全”字符串，结果仍是“安全的”。而如果传入不安全的字符

串，必要时 Django 会自动转义。你可以把它的作用理解为“这个过滤器是安全的，不会引入任何不安全的 HTML”。

is_safe 的存在是有原因的，因为有相当多的字符串操作会把 SafeData 对象变回普通的 str 或 unicode 对象，而捕获所有情况十分困难，Django 会在过滤器执行完毕后修复损伤。

假如有个过滤器在输入后面添加字符串 'xx'，因为没有为结果（除了已经呈现的）引入危险的 HTML 字符，所以应该使用 is_safe 标记过滤器：

```
@register.filter(is_safe=True)
def add_xx(value):
    return '%sxx' % value
```

在启用自动转义的模板中使用这个过滤器时，Django 会转义未标记为“安全”的输入。is_safe 默认为 False，因此不需要标记为安全的过滤器可以省略这个旗标。一定要谨慎，确保安全的字符串真的是安全的。注意，删除字符时可能无意中在结果中留下错乱的 HTML 标签或实体。

例如，删除 > 后可能导致输出中的 <a> 变成 <a，此时为了避免出问题，应该转义输出。类似地，删除分号（;）可能导致 & 变成 &amp，这不是有效的 HTML 实体，应该转义。多数情况下不会这么麻烦，但是审查代码时要留意一下。

把过滤器标记为“安全的”之后，过滤器返回的值会强制转换成字符串。如果过滤器返回布尔值或其他字符串之外的值，标记为安全可能导致意料之外的后果（例如把布尔值 False 转换成字符串 'False'）。

第二种情况没有把输出标记为安全的，HTML 标记不会转义，因此你要自己动手处理。若想把输出标记为安全的字符串，使用 django.utils.safestring.mark_safe()。

不过要小心，你要做的不仅是把输出标记为安全的，而要确保输出真的是安全的。此时，具体怎么做取决于有没有启用自动转义。

编写过滤器时要确保不管有没有启用自动转义都能在模板中正常使用，从而解放模板编写人员。

为了让过滤器知道当前的自动转义状态，注册过滤器函数时把 needs_autoescape 旗标设为 True。（如果不指定，这个旗标默认为 False。）这个旗标告诉 Django，你的过滤器函数要传入一个额外的关键字参数，名为 autoescape，在启用自动转义时值为 True，否则为 False。

例如，下述过滤器突出显示字符串的第一个字符：

```
from django import template
from django.utils.html import conditional_escape
from django.utils.safestring import mark_safe

register = template.Library()

@register.filter(needs_autoescape=True)
def initial_letter_filter(text, autoescape=None):
    first, other = text[0], text[1:]
    if autoescape:
        esc = conditional_escape
    else:
        esc = lambda x: x
    result = '<strong>%s</strong>%s' % (esc(first), esc(other))
    return mark_safe(result)
```

我们指定了 needs_autoescape 旗标和 autoescape 关键字参数，因此调用这个过滤器时知道有没有启用自动转

义。我们通过 autoescape 判断要不要把输入数据传给 django.utils.html.conditional_escape。（在后一种情况下，直接把函数自身当做“转义”函数。）

conditional_escape() 函数的作用与 escape() 类似，不过它只转义不是 SafeData 实例的输入。如果把 SafeData 实例传给 conditional_escape()，数据原封不动地返回。

最后，在上述示例中，我们把结果标记为安全的，因此 HTML 直接插入模板，不再转义。这里无需担心 is_safe 旗标（不过加上也没什么损失）。只要自行处理了自动转义问题，返回安全的字符串，is_safe 旗标就不会再做处理。

过滤器和时区

自定义处理 datetime 对象的过滤器时，注册时通常要把 expects_localtime 旗标设为 True：

```
@register.filter(expects_localtime=True)
def businesshours(value):
    try:
        return 9 <= value.hour < 17
    except AttributeError:
        return ''
```

这样设定之后，如果过滤器的第一个参数是涉及时区的日期时间，根据模板中的时区转换规则，必要时 Django 会先把它转换成当前时区，然后再传给过滤器。

> **复用内置过滤器时避免 XSS 漏洞**
>
> 复用 Django 内置的过滤器时要小心，要把 autoescape=True 传给过滤器，获得正确的自动转义行为，并且避免跨站脚本漏洞。假如你想编写一个 urlize_and_linebreaks 过滤器，把 urlize 和 linebreaksbr 两个过滤器的功能合并到一起，应该这么编写：
>
> ```
> from django.template.defaultfilters import linebreaksbr, urlize
>
> @register.filter
> def urlize_and_linebreaks(text):
>     return linebreaksbr(urlize(text, autoescape=True), autoescape=True)
> ```
>
> 这样，{{ comment|urlize_and_linebreaks }} 等效于 {{ comment|urlize|linebreaksbr }}。

8.7.2 自定义模板标签

标签能做任何事情，比过滤器复杂。Django 提供了一些快捷方式，简化了多数标签类型的编写。首先，我们将探讨这些快捷方式，然后说明在快捷方式不够用时如何从头开始编写标签。

简单的标签

很多模板标签接受几个参数（字符串或模板变量），对输入参数和一些外部信息做些处理之后返回一个结果。

例如有个 current_time 标签，它接受一个格式字符串，返回格式化后的时间字符串。为了简化这种标签的创建，Django 提供了一个辅助函数，simple_tag。它是 django.template.Library 中的一个方法，其参数是一个接受任意个参数的函数，把它包装在 render 函数中之后，再做些前述的必要处理，最后注册到模板系统中。

据此，current_time 函数可以这么编写：

```
import datetime
from django import template

register = template.Library()

@register.simple_tag
def current_time(format_string):
    return datetime.datetime.now().strftime(format_string)
```

关于 simple_tag 辅助函数有几点要注意：

- 调用标签函数时已经检查了必要参数的数量，因此无需我们检查。
- 参数两侧的引号（如果有的话）已经去掉了，接收到的是普通的字符串。
- 如果参数是模板变量，传给标签函数的是变量的当前值，而不是变量本身。

如果模板标签需要访问当前上下文，注册标签时指定 takes_context 参数：

```
@register.simple_tag(takes_context=True)
def current_time(context, format_string):
    timezone = context['timezone']
    return your_get_current_time_method(timezone, format_string)
```

注意，第一个参数的名称必须是 context。takes_context 选项的工作机制参见“引入标签”。如果想为标签起个别的名称，指定 name 参数：

```
register.simple_tag(lambda x: x - 1, name='minusone')

@register.simple_tag(name='minustwo')
def some_function(value):
    return value - 2
```

使用 simple_tag 装饰的函数可以接受任意个位置参数和关键字参数。例如：

```
@register.simple_tag
def my_tag(a, b, *args, **kwargs):
    warning = kwargs['warning']
    profile = kwargs['profile']
    ...
    return ...
```

然后在模板中可以把任意个参数（以空格分开）传给这个标签。与 Python 代码一样，关键字参数的值使用等号（=）设定，而且必须放在位置参数后面。例如：

```
{% my_tag 123 "abcd" book.title warning=message|lower profile=user.profile %}
```

引入标签

另一种常见的模板标签用于渲染另一个模板。例如，Django 的管理后台使用自定义标签在“添加/修改”表单页面下部显示一些按钮。这些按钮的外观相同，但是链接目标根据所编辑的对象有所不同。因此，这些按钮特别适合做成一个小模板，然后使用当前对象填充细节。（管理后台使用的是 submit_row 标签。）

这种标签叫做引入标签（inclusion tag）。这种标签最好通过实例说明。下面我们来编写一个标签，生成指定

Author 对象名下的图书列表。我们将像这样使用这个标签：

```
{% books_for_author author %}
```

得到的结果如下：

```
<ul>
    <li>The Cat In The Hat</li>
    <li>Hop On Pop</li>
    <li>Green Eggs And Ham</li>
</ul>
```

首先，要定义一个接受参数的函数，生成一个字典，为结果提供数据。注意，我们只需返回一个字典，而不是其他复杂的数据结构。返回的字典在模板片段的上下文中使用。

```
def books_for_author(author):
    books = Book.objects.filter(authors__id=author.id)
    return {'books': books}
```

然后，创建用于渲染标签输出的模板。对这个标签来说，模板十分简单：

```
<ul>
{% for book in books %}
    <li>{{ book.title }}</li>
{% endfor %}
</ul>
```

最后，在 Library 对象上调用 inclusion_tag() 方法，创建并注册这个引入标签。这里，如果上述模板保存在模板加载器能搜索到的一个目录中，而且那个文件名为 book_snippet.html，我们可以使用下述代码注册这个标签：

```
# 与前面一样，这里的 register 是 django.template.Library 实例
@register.inclusion_tag('book_snippet.html')
def show_reviews(review):
    ...
```

此外，还可以在创建这个函数时使用 django.template.Template 实例注册引入标签：

```
from django.template.loader import get_template
t = get_template('book_snippet.html')
register.inclusion_tag(t)(show_reviews)
```

有时，引入标签可能需要大量参数，导致模板编写人要记住所需的参数及其顺序。为了避免这种痛苦，Django 为引入标签提供了 takes_context 选项。如果在创建引入标签时指定了这个选项，标签就没有必须的参数了，底层的 Python 函数只有一个参数——调用标签时的模板上下文。假如你编写的一个引入标签始终在包含 home_link 和 home_title 变量的上下文中使用，用于指向主页。此时，Python 函数可以这样编写：

```
@register.inclusion_tag('link.html', takes_context=True)
def jump_link(context):
    return {
        'link': context['home_link'],
        'title': context['home_title'],
    }
```

（注意，函数的第一个参数必须名为 context。）link.html 模板可能包含下述内容：

```
Jump directly to <a href="{{ link }}">{{ title }}</a>.
```

想使用这个自定义标签时，只需加载所在的库，然后不传入参数调用，如下所示：

```
{% jump_link %}
```

注意，指定了 takes_context=True 之后，无需再向模板标签传递参数，它始终能访问上下文。takes_context 参数的默认值是 False。设为 True 时，把上下文对象传给标签，如上例所示。这个示例与前面那个示例唯一的区别就在这里。与简单标签一样，引入标签函数也可以接受任意个位置参数或关键字参数。

赋值标签

为了简化创建为上下文中的变量设值的标签，Django 提供了一个辅助函数，名为 assignment_tag。这个函数的作用与 simple_tag() 类似，不过标签的结果存储在指定的上下文变量中，而不直接输出。前面所举的 current_time 函数可以改写成这样：

```
@register.assignment_tag
def get_current_time(format_string):
    return datetime.datetime.now().strftime(format_string)
```

在模板中，可以使用 as 参数把结果存储在一个变量中，然后在适合的地方输出：

```
{% get_current_time "%Y-%m-%d %I:%M %p" as the_time %}
<p>The time is {{ the_time }}.</p>
```

8.8 自定义模板标签的高级方式

有时，这些自定义模板标签的方式不够用。为了让你能从头开始构建模板标签，Django 提供了模板系统的完整内部细节。

8.8.1 概览

模板系统的运作分为两步：编译和渲染。因此，自定义模板要指定如何编译和渲染。Django 编译模板时，把原始模板分解为一个个"节点"。节点是 django.template.Node 实例，有一个 render() 方法。编译好的模板其实就是一些 Node 对象。

在编译好的模板对象上调用 render() 方法时，模板在节点列表中的各个 Node 对象上调用 render() 方法，并传入指定的上下文。最后，把各个节点的输出拼接在一起，组成模板的输出。因此，自定义模板标签时，要指定如何把原始模板转换成 Node 对象（编译），并且指定节点的 render() 方法要做什么（渲染）。

8.8.2 编写编译函数

模板解析器遇到模板标签时，调用一个 Python 函数，并且传入标签的内容和解析器对象自身。这个函数负责根据标签的内容返回一个 Node 实例。下面我们将从头开始实现前面那个简单的模板标签 {% current_time %}，根据参数中以 strftime() 句法指定的格式显示当前日期和时间。开始行动之前，最好确定标签的句法。这里，假设将像下面这样使用这个标签：

```
<p>The time is {% current_time "%Y-%m-%d %I:%M %p" %}.</p>
```

这个函数的解析器应该获取参数，然后创建一个 Node 对象：

```
from django import template
```

```
def do_current_time(parser, token):
    try:
      tag_name, format_string = token.split_contents()
    except ValueError:
      raise template.TemplateSyntaxError("%r tag requires a single argument"
                                          % token.contents.split()[0])

    if not (format_string[0] == format_string[-1] and format_string[0] in ('"', "'")):
        raise template.TemplateSyntaxError("%r tag's argument should be in quotes"
                                            % tag_name)
    return CurrentTimeNode(format_string[1:-1])
```

注意：

- parser 是模板解析器对象。这里用不到。
- token.contents 是标签的原始内容字符串。这里是 'current_time "%Y-%m-%d %I:%M %p"'。
- token.split_contents() 在空格处把标签的名称和参数分开，不过放在引号内的字符串保持不动。token.contents.split() 简单一些，但是不够可靠，它会在所有空格处拆分，包括引号内的空格。最好始终使用 token.split_contents()。
- 遇到句法错误时，这个函数会抛出 django.template.TemplateSyntaxError，并且提供有用的消息。
- TemplateSyntaxError 异常用到了 tag_name 变量。别在错误消息中硬编码标签的名称，避免标签的名称与函数耦合。token.contents.split()[0] 的值始终是标签的名称，即使标签没有参数。
- 这个函数返回一个 CurrentTimeNode 对象，它具有节点需要知道的关于这个标签的一切信息。这里，它知道的是 "%Y-%m-%d %I:%M %p"。format_string[1:-1] 把标签参数两侧的引号去掉。
- 解析是非常低层的操作。Django 的开发者试过以这个解析系统为基础编写一个小框架，使用 EBNF 语法等技术，但是结果发现模板引擎速度太慢。放在低层是因为那里速度最快。

8.8.3 编写渲染器

自定义标签的第二步是定义一个具有 render() 方法的 Node 子类。接着上面的示例，我们要定义 CurrentTimeNode 类：

```
import datetime
from django import template

class CurrentTimeNode(template.Node):
    def __init__(self, format_string):
        self.format_string = format_string

    def render(self, context):
        return datetime.datetime.now().strftime(self.format_string)
```

注意：

- __init__() 从 do_current_time() 中获取 format_string。节点的选项或参数都通过 __init__() 传递。
- 具体的工作在 render() 方法中做。
- 一般来说，render() 应该静默问题，尤其是在 DEBUG 和 TEMPLATE_DEBUG 设为 False 的生产环境。然而，有些情况下，尤其是 TEMPLATE_DEBUG 设为 True 时，render() 方法可以抛出异常，以便调试。例如，有几个内置的标签在收到错误的参数数量或类型时抛出 django.template.TemplateSyntaxError。

这种编译和渲染的解耦得到的是高效的模板系统，因为无需多次解析一个模板就可以渲染多个上下文。

8.8.4 自动转义方面的注意事项

模板标签的输出不会自动转义。此外，编写模板标签时还有几件事要留意。如果 render() 方法把结果存储在上下文变量中（而不是返回一个字符串），你要负责在适当的时候调用 mark_safe()。最终渲染上下文变量时，会受到彼时的自动转义状态影响，因此无需进一步转义的安全内容应该标记为安全的。

此外，如果模板标签创建新的上下文，用于执行旁支渲染，要把自动转义状态设为当前上下文中的值。Context 类的 __init__ 方法有个名为 autoescape 的参数，它的作用就是如此。例如：

```
from django.template import Context

def render(self, context):
    # ...
    new_context = Context({'var': obj}, autoescape=context.autoescape)
    # ... 处理 new_context ...
```

这种情况不是十分常见，但是自己渲染模板时用得到。例如：

```
def render(self, context):
    t = context.template.engine.get_template('small_fragment.html')
    return t.render(Context({'var': obj}, autoescape=context.autoescape))
```

这里，如果忘了把 context.autoescape 传给新的上下文，结果始终会被转义；如此一来，在 {% autoescape off %} 块中的行为可能与预期不符。

8.8.5 线程安全方面的注意事项

解析节点后，可能在节点上多次调用 render 方法。Django 有时在多线程环境中运行，因此一个节点可能同时在不同的上下文中渲染，响应不同的请求。

鉴于此，一定要确保模板标签是线程安全的。为此，一定不能在节点中存储状态信息。例如，Django 提供的内置模板标签 cycle 遍历指定的字符串列表：

```
{% for o in some_list %}
    <tr class="{% cycle 'row1' 'row2' %}>
        ...
    </tr>
{% endfor %}
```

你可能天真地以为 CycleNode 是这么实现的：

```
import itertools
from django import template

class CycleNode(template.Node):
    def __init__(self, cyclevars):
        self.cycle_iter = itertools.cycle(cyclevars)

    def render(self, context):
        return next(self.cycle_iter)
```

但是，假设有两个模板同时渲染上述模板片段：

1. 线程 1 执行第一次迭代，`CycleNode.render()` 返回 `'row1'`
2. 线程 2 执行第一次迭代，`CycleNode.render()` 返回 `'row2'`
3. 线程 1 执行第二次迭代，`CycleNode.render()` 返回 `'row1'`
4. 线程 2 执行第二次迭代，`CycleNode.render()` 返回 `'row2'`

`CycleNode` 是迭代了，但却是全局迭代的，线程 1 和线程 2 各自始终返回相同的值。这显然不是我们想要的行为！

为了解决这种问题，Django 提供了与当前渲染的模板上下文有关的 `render_context`。它的行为与 Python 字典类似，用于存储多次调用 `render` 方法之间的节点状态。下面使用 `render_context` 重构 `CycleNode` 类：

```
class CycleNode(template.Node):
    def __init__(self, cyclevars):
        self.cyclevars = cyclevars

    def render(self, context):
        if self not in context.render_context:
            context.render_context[self] = itertools.cycle(self.cyclevars)
        cycle_iter = context.render_context[self]
        return next(cycle_iter)
```

注意，完全可以把节点的生命周期内不会变化的全局信息存储为一个属性。

对这里的 `CycleNode` 类来说，`cyclevars` 参数在节点实例化之后就不变了，因此不用放到 `render_context` 里。但是，针对当前渲染模板的状态信息，例如 `CycleNode` 当前迭代的内容，应该存储在 `render_context` 中。

8.8.6 注册标签

最后，像 8.7.1 节那样，使用模块的 `Library` 实例注册标签。例如：

```
register.tag('current_time', do_current_time)
```

`tag()` 方法有两个参数：

1. 模板标签的名称，一个字符串。如果留空，使用编译函数的名称。
2. 编译函数，一个 Python 函数（不是函数名称的字符串形式）。

与注册过滤器一样，这个方法还可以作为装饰器使用：

```
@register.tag(name="current_time")
def do_current_time(parser, token):
    ...

@register.tag
def shout(parser, token):
    ...
```

如果不指定 `name` 参数，如上面的第二个示例所示，Django 将使用函数的名称作为标签的名称。

8.8.7 把模板变量传给标签

虽然可以把任意个参数传给模板标签，然后使用 token.split_contents() 分拆，但是参数都拆包成字符串字面量。如果想通过模板标签的参数传入动态内容（模板变量），要做些额外工作。

上述示例把当前时间格式化成一个字符串，然后返回那个字符串。假如我们想传入 DateTimeField 中的日期时间对象，让那个模板标签格式化呢：

```
<p>This post was last updated at {% format_time blog_entry.date_updated "%Y-%m-%d %I:%M %p"
%}.</p>
```

现在，token.split_contents() 返回三个值：

1. 标签的名称 format_time。
2. 字符串 'blog_entry.date_updated'（没有两侧的引号）。
3. 格式字符串 '"%Y-%m-%d %I:%M %p"'。split_contents() 返回的值中包含字符串字面量两侧的引号。

因此，标签要像下面这样定义：

```
from django import template

def do_format_time(parser, token):
    try:
        # split_contents() 不会分拆放在引号里的字符串
        tag_name, date_to_be_formatted, format_string =
        token.split_contents()
    except ValueError:
        raise template.TemplateSyntaxError("%r tag requires exactly
          two arguments" % token.contents.split()[0])
    if not (format_string[0] == format_string[-1] and
          format_string[0] in ('"', "'")):
        raise template.TemplateSyntaxError("%r tag's argument should
          be in quotes" % tag_name)
    return FormatTimeNode(date_to_be_formatted, format_string[1:-1])
```

此外，还要修改渲染器，检索 blog_entry 对象的 date_updated 属性。这一步可以使用 django.template 包中的 Variable() 类实现。

Variable() 类的用法很简单，先使用变量的名称实例化，然后调用 variable.resolve(context)。例如：

```
class FormatTimeNode(template.Node):
    def __init__(self, date_to_be_formatted, format_string):
        self.date_to_be_formatted =
          template.Variable(date_to_be_formatted)
        self.format_string = format_string

    def render(self, context):
        try:
            actual_date = self.date_to_be_formatted.resolve(context)
            return actual_date.strftime(self.format_string)
        except template.VariableDoesNotExist:
            return ''
```

如果在当前上下文中无法把传入的字符串解析成变量，抛出 VariableDoesNotExist 异常。

8.8.8 在上下文中设定变量

上述示例只是输出值。一般来说，设定模板变量的标签比输出值的标签更灵活。这样，模板编写人便可以复用编码标签创建的值。若想在上下文中设定变量，在 render() 方法中像字典那样为上下文中的变量赋值。下面是 CurrentTimeNode 的更新版本，设定 current_time 模板变量，而不输出：

```
import datetime
from django import template

class CurrentTimeNode2(template.Node):
    def __init__(self, format_string):
        self.format_string = format_string
    def render(self, context):
        context['current_time'] =
          datetime.datetime.now().strftime(self.format_string)
        return ''
```

注意，render() 方法返回了一个空字符串。render() 始终应该返回一个字符串。如果模板标签只设定变量，render() 应该返回一个空字符串。这个新版本的用法如下：

```
{% current_time "%Y-%M-%d %I:%M %p" %}
<p>The time is {{ current_time }}.</p>
```

上下文中的变量作用域

上下文中设定的变量只在设定它的模板块中可用。这种行为是故意为之的，目的是为变量提供一个作用域，以防与其他块的上下文冲突。

但是，CurrentTimeNode2 有个问题：变量名称 current_time 是硬编码的。这意味着，模板中不能再使用 {{ current_time }}，因为 {% current_time %} 会毫不客气地覆盖那个变量的值。

正确的做法是让模板标签指定变量的名称，如下所示：

```
{% current_time "%Y-%M-%d %I:%M %p" as my_current_time %}
<p>The current time is {{ my_current_time }}.</p>
```

为此，编译函数和 Node 类都要重构，如下所示：

```
import re

class CurrentTimeNode3(template.Node):
    def __init__(self, format_string, var_name):
        self.format_string = format_string
        self.var_name = var_name
    def render(self, context):
        context[self.var_name] =
          datetime.datetime.now().strftime(self.format_string)
        return ''

def do_current_time(parser, token):
    # 这一版使用正则表达式解析标签的内容
```

```
    try:
        # None 的作用相当于在空格处分析
        tag_name, arg = token.contents.split(None, 1)
    except ValueError:
        raise template.TemplateSyntaxError("%r tag requires arguments"
          % token.contents.split()[0])
    m = re.search(r'(.*?) as (\w+)', arg)
    if not m:
        raise template.TemplateSyntaxError("%r tag had invalid arguments" % tag_name)
    format_string, var_name = m.groups()
    if not (format_string[0] == format_string[-1] and format_string[0] in ('"', "'")):
        raise template.TemplateSyntaxError("%r tag's argument should be in quotes"
          % tag_name)
    return CurrentTimeNode3(format_string[1:-1], var_name)
```

这一版不同的地方是，`do_current_time()` 获取的格式字符串和变量名称都传给 `CurrentTimeNode3`。最后，如果想让更新上下文的模板标签具有简单的句法，可以使用前文介绍的辅助标签。

8.8.9 一直解析到另一个块标签

模板标签可以配对。例如，内置的 `{% comment %}` 标签把 `{% endcomment %}` 之前的内容都隐藏起来。如果想创建这种模板标签，在编译函数中使用 `parser.parse()`。下面是 `{% comment %}` 标签的简单实现：

```
def do_comment(parser, token):
    nodelist = parser.parse(('endcomment',))
    parser.delete_first_token()
    return CommentNode()

class CommentNode(template.Node):
    def render(self, context):
        return ''
```

> **提示**
>
> `{% comment %}` 标签的真正实现方式稍有不同，允许 `{% comment %}` 和 `{% endcomment %}` 之间有不可用的模板标签。为此，`parser.delete_first_token()` 前面调用的是 `parser.skip_past('endcomment')`，而非 `parser.parse(('endcomment',))`，以免生成节点列表。

`parser.parse()` 的参数是一个元组，指定要“解析到的”块标签。它的返回值是一个 `django.template.NodeList` 实例，这是解析器在遇到那个元组中的标签名称之前得到的 Node 对象列表。对上述示例中的 `nodelist = parser.parse(('endcomment',))` 来说，得到的节点列表包含 `{% comment %}` 和 `{% endcomment %}` 之间的所有节点，但是不含 `{% comment %}` 和 `{% endcomment %}` 自身。

调用 `parser.parse()` 之后，解析器尚未解析 `{% endcomment %}` 标签，因此要调用 `parser.delete_first_token()`。`CommentNode.render()` 只是返回一个空字符串，即 `{% comment %}` 和 `{% endcomment %}` 之间的一切都被忽略。

8.8.10 一直解析到另一个块标签，并且保存内容

在上述示例中，`do_comment()` 丢掉 `{% comment %}` 和 `{% endcomment %}` 之间的一切内容。但是有时可能需要对一对块标签中的内容做些处理。例如，下述自定义模板标签 `{% upper %}` 把它和 `{% endupper %}` 之间的内

容变成大写：

```
{% upper %}This will appear in uppercase, {{ your_name }}.{% endupper %}
```

与前面的示例一样，我们要使用 `parser.parse()`。但是这一次要把得到的节点列表传给 `Node` 子类：

```
def do_upper(parser, token):
    nodelist = parser.parse(('endupper',))
    parser.delete_first_token()
    return UpperNode(nodelist)

class UpperNode(template.Node):
    def __init__(self, nodelist):
        self.nodelist = nodelist

    def render(self, context):
        output = self.nodelist.render(context)
        return output.upper()
```

这里只有一处新内容：`UpperNode.render()` 中的 `self.nodelist.render(context)`。如果想查找复杂渲染的更多示例，请看 `django/template/defaulttags.py` 中 `{% for %}` 标签的源码，以及 `django/template/smartif.py` 中 `{% if %}` 标签的源码。

8.9 接下来

下一章继续讨论高级话题，说明 Django 模型的高级用法。

第 9 章 Django 模型的高级用法

第 4 章介绍了 Django 的数据库层，说明了如何定义模型，以及如何使用数据库 API 创建、检索、更新和删除记录。本章将介绍 Django 数据库层的一些高级特性。

9.1 相关的对象

我们在第 4 章定义了下面几个模型：

```
from django.db import models

class Publisher(models.Model):
    name = models.CharField(max_length=30)
    address = models.CharField(max_length=50)
    city = models.CharField(max_length=60)
    state_province = models.CharField(max_length=30)
    country = models.CharField(max_length=50)
    website = models.URLField()

    def __str__(self):
        return self.name

class Author(models.Model):
    first_name = models.CharField(max_length=30)
    last_name = models.CharField(max_length=40)
    email = models.EmailField()

    def __str__(self):
        return '%s %s' % (self.first_name, self.last_name)

class Book(models.Model):
    title = models.CharField(max_length=100)
    authors = models.ManyToManyField(Author)
    publisher = models.ForeignKey(Publisher)
    publication_date = models.DateField()

    def __str__(self):
        return self.title
```

我们知道，访问数据库中某一列很简单，只需访问对象的属性。例如，若想查看 ID 为 50 的图书的书名，可以这么做：

```
>>> from mysite.books.models import Book
>>> b = Book.objects.get(id=50)
>>> b.title
'The Django Book'
```

但是前面没有指出，相关的对象（ForeignKey 或 ManyToManyField 字段）行为稍微有些不同。

9.1.1 访问外键值

访问 ForeignKey 类型的字段时，得到的是相关的模型对象。例如：

```
>>> b = Book.objects.get(id=50)
>>> b.publisher
<Publisher: Apress Publishing>
>>> b.publisher.website
'http://www.apress.com/'
```

ForeignKey 字段也能反向使用，不过因为关系是不对称的，行为稍有不同。若想获取指定出版社出版的所有图书，要使用 publisher.book_set.all()，如下所示：

```
>>> p = Publisher.objects.get(name='Apress Publishing')
>>> p.book_set.all()
[<Book: The Django Book>, <Book: Dive Into Python>, ...]
```

其实，book_set 就是一个 QuerySet 对象（参见第 4 章），可以过滤和切片。例如：

```
>>> p = Publisher.objects.get(name='Apress Publishing')
>>> p.book_set.filter(title__icontains='django')
[<Book: The Django Book>, <Book: Pro Django>]
```

book_set 属性是生成的：把模型名的小写形式与 _set 连在一起。

9.1.2 访问多对多值

多对多值与外键值的获取方式类似，不过处理的是 QuerySet 值，而非模型实例。例如，查看一本的的作者要这么做：

```
>>> b = Book.objects.get(id=50)
>>> b.authors.all()
[<Author: Adrian Holovaty>, <Author: Jacob Kaplan-Moss>]
>>> b.authors.filter(first_name='Adrian')
[<Author: Adrian Holovaty>]
>>> b.authors.filter(first_name='Adam')
[]
```

反过来也可以。如果想查看一位作者撰写的所有图书，使用 author.book_set，如下所示：

```
>>> a = Author.objects.get(first_name='Adrian',
last_name='Holovaty')
>>> a.book_set.all()
[<Book: The Django Book>, <Book: Adrian's Other Book>]
```

与 ForeignKey 字段一样，这里的 book_set 也是生成的：把模型名的小写形式与 _set 连在一起。

9.2 管理器

在 Book.objects.all() 语句中，objects 是个特殊的属性，我们通过它查询数据库。第 4 章简单说过，这是模型的*管理器*（manager）。现在，我们要深入说明管理器的作用和用法。

简单来说，模型的管理器是 Django 模型用于执行数据库查询的对象。一个 Django 模型至少有一个管理器，而且可以自定义管理器，定制访问数据库的方式。自定义管理器可能出于两方面的原因：添加额外的管理器方法和（或）修改管理器返回的 QuerySet。

9.2.1 添加额外的管理器方法

添加额外的管理器方法是为模型添加数据表层功能的首选方式。（数据行层的功能，即在模型对象的单个实例上执行的操作，使用模型方法。本章后面将做说明。）

举个例子。下面我们为 Book 模型添加一个管理器方法 title_count()，它的参数是一个关键字，返回书名中包含关键字的图书数量。（这个示例是故意设计出来的，不过能说明管理器的运作方式。）

```
# models.py

from django.db import models

# ... Author 和 Publisher 模型省略了 ...

class BookManager(models.Manager):
    def title_count(self, keyword):
        return self.filter(title__icontains=keyword).count()

class Book(models.Model):
    title = models.CharField(max_length=100)
    authors = models.ManyToManyField(Author)
    publisher = models.ForeignKey(Publisher)
    publication_date = models.DateField()
    num_pages = models.IntegerField(blank=True, null=True)
    objects = BookManager()

    def __str__(self):
        return self.title
```

这段代码有几点需要注意：

1. 我们定义的 BookManager 类扩展 django.db.models.Manager。类中只有一个方法，title_count()，做相关的计算。注意，这个方法使用了 self.filter()，其中 self 指代管理器自身。
2. 我们把 BookManager() 赋值给模型的 objects 属性。这么做的效果是替换模型的“默认”管理器，即未指定管理器时自动创建的 objects。我们仍把它叫做 objects，以便与自动创建的管理器保持一致。

创建好管理器之后，可以像下面这样使用：

```
>>> Book.objects.title_count('django')
4
>>> Book.objects.title_count('python')
18
```

显然，这只是示例，如果你在自己的交互式解释器中输入上述代码，得到的返回值可能不同。

我们为什么想要添加 title_count() 这样的方法呢？为的是封装经常执行的查询，以免代码重复。

9.2.2 修改管理器返回的查询集合

管理器的基本查询集合返回系统中的所有对象。例如，`Book.objects.all()` 返回数据库中的所有图书。若想覆盖管理器的基本查询集合，覆盖 `Manager.get_queryset()` 方法。`get_queryset()` 方法应该返回一个 `QuerySet`，包含所需的属性。

例如，下述模型有两个管理器，一个返回所有对象，另一个只返回 Roald Dahl 写的书。

```
from django.db import models

# 首先，定义 Manager 子类
class DahlBookManager(models.Manager):
    def get_queryset(self):
        return super(DahlBookManager, self).get_queryset().filter(author='Roald Dahl')

# 然后，放入 Book 模型
class Book(models.Model):
    title = models.CharField(max_length=100)
    author = models.CharField(max_length=50)
    # ...

    objects = models.Manager() # 默认的管理器
    dahl_objects = DahlBookManager() # 专门查询 Dahl 的管理器
```

对这个示例模型来说，`Book.objects.all()` 返回数据库中的所有图书，而 `Book.dahl_objects.all()` 只返回 Roald Dahl 写的书。注意，我们明确地把 `objects` 设为一个普通的 `Manager` 示例，如若不然，唯一可用的管理器将是 `dahl_objects`。当然，`get_queryset()` 返回的是一个 `QuerySet` 对象，因此可以在其上调用 `filter()`、`exclude()` 和其他所有 `QuerySet` 支持的方法。所以，下述语句都是有效的：

```
Book.dahl_objects.all()
Book.dahl_objects.filter(title='Matilda')
Book.dahl_objects.count()
```

这个示例还指出了另一个有用的技术：在同一个模型上使用多个管理器。只要需要，可以为模型添加任意个 `Manager()` 实例。这么做，可以轻易为模型定义常用的“过滤器”。例如：

```
class MaleManager(models.Manager):
    def get_queryset(self):
        return super(MaleManager, self).get_queryset().filter(sex='M')

class FemaleManager(models.Manager):
    def get_queryset(self):
        return super(FemaleManager, self).get_queryset().filter(sex='F')

class Person(models.Model):
    first_name = models.CharField(max_length=50)
    last_name = models.CharField(max_length=50)
    sex = models.CharField(max_length=1,
                           choices=(
                                    ('M', 'Male'),
                                    ('F', 'Female')
                                    )
                           )
```

```
    people = models.Manager()
    men = MaleManager()
    women = FemaleManager()
```

这样定义之后，可以使用 `Person.men.all()`、`Person.women.all()` 和 `Person.people.all()`，而且能得到预期的结果。自定义 `Manager` 对象时要注意，Django 遇到的第一个管理器（按照在模型中定义的顺序）有特殊的状态。Django 把它解释的第一个管理器定义为"默认的"管理器，而且 Django 在很多地方（管理后台不在此列）只使用那个管理器。

鉴于此，通常最好小心选择默认的管理器，以防把 `get_queryset()` 返回的结果覆盖掉，无法检索所需的对象。

9.3 模型方法

模型中自定义的方法为对象添加数据行层的功能。管理器的作用是执行数据表层的操作，而模型方法处理的是具体的模型实例。这个技术的价值很大，能把业务逻辑统一放在一个地方，即模型中。

通过示例说明最简单。下述模型有一个自定义的方法：

```
from django.db import models

class Person(models.Model):
    first_name = models.CharField(max_length=50)
    last_name = models.CharField(max_length=50)
    birth_date = models.DateField()

    def baby_boomer_status(self):
        # 返回一个人的出生日期与婴儿潮的关系
        import datetime
        if self.birth_date < datetime.date(1945, 8, 1):
            return "Pre-boomer"
        elif self.birth_date < datetime.date(1965, 1, 1):
            return "Baby boomer"
        else:
            return "Post-boomer"

    def _get_full_name(self):
        # 返回一个人的全名
        return '%s %s' % (self.first_name, self.last_name)

    full_name = property(_get_full_name)
```

各个模型自动具有的方法列表参见附录 A。这些方法基本上都可以覆盖（参见下文），其中几个最常定义：

- `__str__()`。这是 Python 的一个"魔法方法"，返回对象的 Unicode 表示形式。需要以普通的字符串显示模型实例时，Python 和 Django 会调用这个方法。尤其要注意，在交互式控制台或管理后台中显示对象调用的都是这个方法。这个方法一定要自定义，因为默认的实现没什么用。
- `get_absolute_url()`。这个方法告诉 Django 如何计算一个对象的 URL。Django 在管理后台和需要生成对象的 URL 时调用这个方法。具有唯一标识的 URL 的对象都要定义这个方法。

9.3.1 覆盖预定义的模型方法

还有一系列封装数据库行为的模型方法有时也需要自定义。尤其是 save() 和 delete()，经常需要修改它们的运作方式。这些方法（以及任何其他模型方法）允许覆盖，让你调整它们的行为。覆盖内置方法的一个典型使用场景是在保存对象时做些事情。例如（save() 方法接受的参数参见文档）：

```
from django.db import models

class Blog(models.Model):
    name = models.CharField(max_length=100)
    tagline = models.TextField()

    def save(self, *args, **kwargs):
        do_something()
        super(Blog, self).save(*args, **kwargs) # 调用“真正的”save () 方法
        do_something_else()
```

也可以在特定的条件下禁止保存：

```
from django.db import models

class Blog(models.Model):
    name = models.CharField(max_length=100)
    tagline = models.TextField()

    def save(self, *args, **kwargs):
        if self.name == "Yoko Ono's blog":
            return # Yoko 肯定不会开博客的!
        else:
            super(Blog, self).save(*args, **kwargs) # 调用“真正的”save () 方法
```

一定要记得调用超类中的方法，即 super(Blog, self).save(*args, **kwargs)，确保把对象保存到数据库中。如果忘记，默认的行为不会发生，根本不会触及数据库。

此外，还要记得传递传给模型方法的参数，即 *args, **kwargs。Django 时常扩展内置模型方法的能力，为其添加新参数。在方法定义中使用 *args, **kwargs 能确保代码自动支持未来添加的参数。

9.4 执行原始 SQL

模型的查询 API 不够用时，可以编写原始 SQL。Django 为执行原始 SQL 查询提供了两种方式：使用 Manager.raw() 执行，返回模型实例集合；或者完全不用模型层，直接执行自定义的 SQL。

> **提醒**
>
> 编写原始 SQL 时要非常小心。一定要正确转义通过 params 传入的参数，以防 SQL 注入攻击。

9.5 执行原始查询

管理器的 raw() 方法用于执行原始的 SQL 查询，其返回结果是模型实例集合：

```
Manager.raw(raw_query, params=None, translations=None)
```

这个方法的参数是一个原始的 SQL 查询，执行后返回一个 django.db.models.query.RawQuerySet 实例。RawQuerySet 实例可以像常规的 QuerySet 对象一样迭代，获取里面的模型对象。下面通过一个示例说明。假设有下述模型：

```
class Person(models.Model):
    first_name = models.CharField(...)
    last_name = models.CharField(...)
    birth_date = models.DateField(...)
```

可以像这样执行 SQL 查询：

```
>>> for p in Person.objects.raw('SELECT * FROM myapp_person'):
...     print(p)
John Smith
Jane Jones
```

当然，这个示例没什么让人兴奋的，与 Person.objects.all() 的效果一样。然而，raw() 有些特别强大的功能。

9.5.1 模型对应的表名

上例中的 Person 表名是怎么来的呢？Django 默认把"应用标注"（manage.py startapp 命令指定的名称）与类名使用下划线联结在一起得到数据库表名。在上述示例中，我们假设 Person 模型在 myapp 应用中，因此对应的表是 myapp_person。

更多信息参见 db_table 选项的文档，通过它还可以自定义数据表的名称。

提醒

Django 不检查传给 raw() 方法的 SQL 语句，而是预期查询能返回一系列数据库行，但不做任何强制措施。如果不返回一系列行，抛出错误（可能晦涩难懂）。

9.5.2 把查询中的字段映射到模型字段上

raw() 自动把查询中的字段映射到模型字段上。查询中的字段顺序无关紧要。也就是说，下述两个查询的作用完全一样：

```
>>> Person.objects.raw('SELECT id, first_name, last_name, birth_date FROM myapp_person')
...
>>> Person.objects.raw('SELECT last_name, birth_date, first_name, id FROM myapp_person')
...
```

二者之间通过名称匹配。这意味着，可以使用 SQL 的 AS 子句把查询中的字段映射到模型字段上。因此，如果其他表中有 Person 数据，可以轻易将其映射为 Person 实例：

```
>>> Person.objects.raw('''SELECT first AS first_name,
...                              last AS last_name,
...                              bd AS birth_date,
...                              pk AS id,
...                       FROM some_other_table''')
```

只要名称匹配就能正确创建模型实例。此外，还可以使用 raw() 方法的 translations 参数把查询中的字段映

射到模型字段上。这个参数的值是一个字典，把查询中的字段名称映射到模型字段的名称上。例如，上述查询还可以写成：

```
>>> name_map = {'first': 'first_name', 'last': 'last_name', 'bd': 'birth_date', 'pk': 'id'}
>>> Person.objects.raw('SELECT * FROM some_other_table', translations=name_map)
```

9.5.3 索引查找

raw() 支持索引，因此如果只想获得第一个结果，可以这样写：

```
>>> first_person = Person.objects.raw('SELECT * FROM myapp_person')[0]
```

然而，索引和切片不是在数据库层执行的。如果数据库中的 Person 对象很多，最好在 SQL 查询中限制数量：

```
>>> first_person = Person.objects.raw('SELECT * FROM myapp_person LIMIT 1')[0]
```

9.5.4 延期模型字段

还可以把字段排除在外：

```
>>> people = Person.objects.raw('SELECT id, first_name FROM myapp_person')
```

这个查询返回的是延期的 Person 对象（参阅 defer()）。这意味着，查询排除的字段将按需加载。例如：

```
>>> for p in Person.objects.raw('SELECT id, first_name FROM myapp_person'):
...     print(p.first_name, # 这个属性由查询取回
...           p.last_name) # 这个属性按需取回
...
John Smith
Jane Jones
```

从表面看，好像这个查询把名字和姓都取回了。然而，这个示例其实发起了三个查询。raw() 执行的查询只取回名字，两人的姓在打印时按需取回。

只有一个字段是不能排除的——主键字段。Django 使用主键标识模型实例，因此必须始终包含在原始查询中。忘记主键时，抛出 InvalidQuery 异常。

9.5.5 添加注解

执行的查询还可以包含模型中没有定义的字段。例如，可以使用 PostgreSQL 的 age() 函数让数据库计算所得诸人的年龄：

```
>>> people = Person.objects.raw('SELECT *, age(birth_date) AS age FROM myapp_person')
>>> for p in people:
...     print("%s is %s." % (p.first_name, p.age))
John is 37.
Jane is 42.
...
```

9.5.6 为 raw() 传递参数

如果想执行参数化查询，可以把 params 参数传给 raw()：

```
>>> lname = 'Doe'
>>> Person.objects.raw('SELECT * FROM myapp_person WHERE last_name = %s', [lname])
```

params 的值是一个参数列表或字典。使用列表时，查询字符串中的占位符是 %s；使用字典时，占位符是 %(key)s（当然，其中的 key 要替换成字典的键）——不管使用何种数据库引擎，都是如此。这些占位符会替换成 params 参数中的值。

> **原始查询中不要使用字符串格式化！**
>
> 上述查询可能会错误地写成：
>
> ```
> >>> query = 'SELECT * FROM myapp_person WHERE last_name = %s' % lname
> Person.objects.raw(query)
> ```
>
> 千万别这么写！
>
> 使用 params 参数能完全避免 SQL 注入攻击。这是一种常见的漏洞，攻击者能设法向数据库中注入任意的 SQL。如果使用字符串插值，迟早有一天你会变成 SQL 注入的牺牲者。记住，一定要使用 params 参数，这样便能得到保护。

9.6 直接执行自定义的 SQL

有时，连 Manager.raw() 可能都不够用，例如执行的查询不完全映射到模型上，或者直接执行 UPDATE、INSERT 或 DELETE 查询。此时，完全可以直接访问数据库，绕开模型层。django.db.connection 对象表示默认的数据库连接。若想使用这个数据库连接，调用 connection.cursor()，获取一个游标对象。然后，调用 cursor.execute(sql, [params]) 执行 SQL，再调用 cursor.fetchone() 或 cursor.fetchall() 返回所得的行。例如：

```
from django.db import connection

def my_custom_sql(self):
    cursor = connection.cursor()
    cursor.execute("UPDATE bar SET foo = 1 WHERE baz = %s", [self.baz])
    cursor.execute("SELECT foo FROM bar WHERE baz = %s", [self.baz])
    row = cursor.fetchone()

    return row
```

注意，传入参数时，如果查询中有百分号，应该编写两个百分号：

```
cursor.execute("SELECT foo FROM bar WHERE baz = '30%'")
cursor.execute("SELECT foo FROM bar WHERE baz = '30%%' AND
  id = %s", [self.id])
```

使用多个数据库时，可以使用 django.db.connections 获取指定数据库的连接（和游标）。django.db.connections 是一个类似字典的对象，可以使用别名取回指定连接：

```
from django.db import connections
cursor = connections['my_db_alias'].cursor()
# 其他代码...
```

默认情况下，Python DB API 返回的结果不带字段名称，因此得到的是列表，而不是字典。在损失少许性能

的前提下，可以像这样返回字典：

```
def dictfetchall(cursor):
    # 把游标中的行以字典的形式返回
    desc = cursor.description
    return [
        dict(zip([col[0] for col in desc], row))
        for row in cursor.fetchall()
    ]
```

二者之间的区别如下例所示：

```
>>> cursor.execute("SELECT id, parent_id FROM test LIMIT 2");
>>> cursor.fetchall()
((54360982L, None), (54360880L, None))

>>> cursor.execute("SELECT id, parent_id FROM test LIMIT 2");
>>> dictfetchall(cursor)
[{'parent_id': None, 'id': 54360982L}, {'parent_id': None, 'id': 54360880L}]
```

9.6.1 连接和游标

`connection` 和 `cursor` 基本上实现了 PEP 249 定义的 Python DB API，不过对事务的处理方式有所不同。如果你不熟悉 Python DB API，要注意，`cursor.execute()` 中的 SQL 语句使用占位符 `%s`，而不直接把参数添加到 SQL 查询中。

使用占位符时，底层数据库库会自动转义参数。还要注意，Django 使用的占位符是 `%s`，而不是 `?`。后者是 SQLite 的 Python 绑定使用的。这样做是为了一致性和健全性。像下面这样把游标当做上下文管理器使用：

```
with connection.cursor() as c:
    c.execute(...)
```

等效于：

```
c = connection.cursor()
try:
    c.execute(...)
finally:
    c.close()
```

9.6.2 添加额外的管理器方法

添加额外的管理器方法是为模型添加数据表层功能的首选方式。（数据行层的功能，即在模型对象的单个实例上执行的操作，使用模型方法。）自定义的管理器方法可以返回任何需要的内容，不一定是 `QuerySet`。

例如，下面这个自定义的管理器提供了 `with_counts()` 方法，它返回所有 `OpinionPoll` 对象，每个对象都有额外的 `num_responses` 属性，其值为聚合查询的结果：

```
from django.db import models

class PollManager(models.Manager):
    def with_counts(self):
        from django.db import connection
```

```
        cursor = connection.cursor()
        cursor.execute("""
            SELECT p.id, p.question, p.poll_date, COUNT(*)
            FROM polls_opinionpoll p, polls_response r
            WHERE p.id = r.poll_id
            GROUP BY p.id, p.question, p.poll_date
            ORDER BY p.poll_date DESC""")
        result_list = []
        for row in cursor.fetchall():
            p = self.model(id=row[0], question=row[1], poll_date=row[2])
            p.num_responses = row[3]
            result_list.append(p)
        return result_list

class OpinionPoll(models.Model):
    question = models.CharField(max_length=200)
    poll_date = models.DateField()
    objects = PollManager()

class Response(models.Model):
    poll = models.ForeignKey(OpinionPoll)
    person_name = models.CharField(max_length=50)
    response = models.TextField()
```

对这个示例来说，要使用 `OpinionPoll.objects.with_counts()` 获取具有 `num_responses` 属性的 `OpinionPoll` 对象列表。还有一点要注意：管理器方法可以访问 `self.model`，获取所依附的模型类。

9.7 接下来

下一章说明 Django 的通用视图框架。遵守常见模式构建网站时，使用通用视图可以节省时间。

第 10 章 通用视图

本书不断强调，往坏了说，Web 开发是单调乏味的。目前，我们介绍了 Django 为了缓和这种单调在模型和模板层上所做的努力，但是 Web 开发者在视图层也会经历这种乏味的体验。

为了降低这方面的痛苦，Django 开发了*通用视图*（generic view）。

通用视图把视图开发中常用的写法和模式抽象出来，让你编写少量代码就能快速实现常见的数据视图。显示对象列表就是这样一种任务。

有了通用视图，可以把模型作为额外的参数传给 URL 配置。Django 自带的通用视图能实现下述功能：

- 列出对象并显示单个对象的详细信息。如果创建一个管理会议的应用程序，那么 `TalkListView` 和 `RegisteredUserListView` 就是列表视图。某一个演讲的页面就是详细信息视图。
- 呈现基于日期的对象，显示为年月日归档页面（带有详细信息），以及“最新”页面。
- 让用户创建、更新和删除对象——需不需要授权都行。

综上，这些视图提供了简单易用的接口，在视图中显示数据库里的数据时能为开发者执行多数常见的任务。然而，显示视图只是 Django 全面的基于类的视图系统的作用之一。Django 提供的其他基于类的视图的完整介绍和详细说明参见附录 C。

10.1 对象的通用视图

在视图中呈现数据库里的内容时最能体现 Django 的通用视图是多么强大。这是一件十分常见的任务，因此 Django 内建了很多这方面的通用视图，不费吹灰之力就能生成对象列表和详细信息视图。

下面举几个例子。我们将使用下述模型：

```python
# models.py
from django.db import models

class Publisher(models.Model):
    name = models.CharField(max_length=30)
    address = models.CharField(max_length=50)
    city = models.CharField(max_length=60)
    state_province = models.CharField(max_length=30)
    country = models.CharField(max_length=50)
    website = models.URLField()

    class Meta:
        ordering = ["-name"]

    def __str__(self):
        return self.name

class Author(models.Model):
```

```python
    salutation = models.CharField(max_length=10)
    name = models.CharField(max_length=200)
    email = models.EmailField()
    headshot = models.ImageField(upload_to='author_headshots')

    def __str__(self):
        return self.name

class Book(models.Model):
    title = models.CharField(max_length=100)
    authors = models.ManyToManyField('Author')
    publisher = models.ForeignKey(Publisher)
    publication_date = models.DateField()
```

然后，定义一个视图：

```python
# views.py
from django.views.generic import ListView
from books.models import Publisher

class PublisherList(ListView):
    model = Publisher
```

最后，把视图与 URL 关联起来：

```python
# urls.py
from django.conf.urls import url
from books.views import PublisherList

urlpatterns = [
    url(r'^publishers/$', PublisherList.as_view()),
]
```

这就是我们需要编写的全部 Python 代码。当然，还要编写模板。我们可以为视图添加一个 `template_name` 属性，明确指明使用哪个模板；如果没明确指定，Django 将从对象的名称中推知。这里，推知的模板是 `books/publisher_list.html`，其中“books”是模型所在应用的名称，“publisher”是模型名的小写形式。

因此，如果 `TEMPLATES` 设置中 `DjangoTemplates` 后台的 `APP_DIRS` 选项设为 `True`，那么这个模板的位置是 `/path/to/project/books/templates/books/publisher_list.html`。

渲染这个模板时，上下文中有个名为 `object_list` 的变量，它的值是所有出版社对象。下面是一个十分简单的模板：

```
{% extends "base.html" %}

{% block content %}
    <h2>Publishers</h2>
    <ul>
        {% for publisher in object_list %}
            <li>{{ publisher.name }}</li>
        {% endfor %}
    </ul>
{% endblock %}
```

以上就是全部代码。通用视图所有的强大功能都通过修改属性实现。附录 C 将详细说明所有通用视图及其选项，还将介绍定制和扩展通用视图的一些常见方式。

10.2 提供“友好的”模板上下文

你可能注意到了，上述出版社列表模板示例把所有出版社存储在一个名为 object_list 的变量中。这样虽然可行，但是对模板编写人不够友好，他们想知道的是处理的是出版社。

如果处理的是模型对象，Django 已经为你提供了这样一个变量。处理对象或查询集合时，Django 将向上下文中添加一个以模型类名小写形式命名的变量。这个变量与 object_list 同时存在，不过所含的数据完全相同。这里，变量名为 publisher_list。

如果这还不够，可以自行设定上下文变量的名称。通用视图的 context_object_name 属性用于指定要使用的上下文变量名：

```
# views.py
from django.views.generic import ListView
from books.models import Publisher

class PublisherList(ListView):
    model = Publisher
    context_object_name = 'my_favorite_publishers'
```

为 context_object_name 设定一个友好的值总是好的，设计模板的同事会感谢你的。

10.3 提供额外的上下文变量

通常，除了通用视图提供的信息之外，还想显示一些额外信息。例如，在各个出版社的详细信息页面显示出版的图书列表。DetailView 通用视图在上下文中提供了出版社信息，但是如何在模板中获取额外的信息呢？

答案是扩展 DetailView，自己实现 get_context_data 方法。默认的实现只为模板提供该显示的对象，不过可以覆盖，提供更多信息：

```
from django.views.generic import DetailView
from books.models import Publisher, Book

class PublisherDetail(DetailView):

    model = Publisher

    def get_context_data(self, **kwargs):
        # 先调用原来的实现，获取上下文
        context = super(PublisherDetail, self).get_context_data(**kwargs)
        # 把所有图书构成的查询集合添加到上下文中
        context['book_list'] = Book.objects.all()
        return context
```

提示

默认情况下，get_context_data 会把所有父类的上下文数据与当前类的合并在一起。如果在调整上下文的子类中不想使用这种行为，要在超类上调用 get_context_data。如果两个类没有在上下文中定义相同的键，这样得到的结果符合预期。但是，如果尝试覆盖超类设定的键（在调用 super 之后），当子类想覆盖所有超类时，子类也必须在调用 super 之后显式设定那个键。

10.4 显示对象子集

现在仔细分析一下我们一直使用的 model 属性。这个属性指定视图操作的数据库模型，在操作单个对象或对象集合的通用视图中都可用。然而，这不是指定视图操作哪些对象的唯一方式，此外还可以使用 queryset 属性指定一组对象：

```
from django.views.generic import DetailView
from books.models import Publisher

class PublisherDetail(DetailView):

    context_object_name = 'publisher'
    queryset = Publisher.objects.all()
```

model = Publisher 其实是 queryset = Publisher.objects.all() 的简洁形式。然而，使用 queryset 可以过滤对象列表，进一步指定要在视图中查看的对象。下面举个列子。我们可能想按照出版日期排序图书列表，把最近出版的放在前面：

```
from django.views.generic import ListView
from books.models import Book

class BookList(ListView):
    queryset = Book.objects.order_by('-publication_date')
    context_object_name = 'book_list'
```

这个示例相当简单，不过却很好地说明了其中的思想。当然，通常你想做的可能不仅仅是重新排序对象。如果想呈现特定出版社出版的图书列表，也可以使用这个技术：

```
from django.views.generic import ListView
from books.models import Book

class AcmeBookList(ListView):

    context_object_name = 'book_list'
    queryset = Book.objects.filter(publisher__name='Acme Publishing')
    template_name = 'books/acme_list.html'
```

注意，除了过滤查询集合之外，我们还自定义了模板名称。如若不然，通用视图会使用显示普通对象列表的模板，而这可能不是你想要的。

还应注意，这不是显示特定出版社旗下图书的优雅方式。如果想添加关于出版社的其他页面，需要在 URL 配置中多添加几行代码，而当出版社变多后，这样做也不合理。下一节将解决这个问题。

10.5 动态过滤

另一个常见的需求是根据 URL 中指定的键过滤列表页面中的对象。前面，我们在 URL 配置中硬编码出版社的名称，但是如果我们想编写一个视图显示随意一家出版社旗下的所有图书呢？

这也很方便，我们可以覆盖 ListView 的 get_queryset() 方法。它的默认实现是返回 queryset 属性的值，不过我们可以添加更多逻辑。这里的关键是，调用基于类的视图时，很多有用的东西存储到 self 中了，除了请求（self.request）之外，还有根据 URL 配置捕获的位置参数（self.args）和关键字参数（self.kwargs）。

下述 URL 配置只有一个捕获组：

```
# urls.py
from django.conf.urls import url
from books.views import PublisherBookList

urlpatterns = [
    url(r'^books/([\w-]+)/$', PublisherBookList.as_view()),
]
```

然后，编写 PublisherBookList 视图类：

```
# views.py
from django.shortcuts import get_object_or_404
from django.views.generic import ListView
from books.models import Book, Publisher

class PublisherBookList(ListView):

    template_name = 'books/books_by_publisher.html'

    def get_queryset(self):
        self.publisher = get_object_or_404(Publisher,  name=self.args[0])
        return Book.objects.filter(publisher=self.publisher)
```

可以看出，为查询集合添加逻辑还是相当容易的。如果需要，可以使用 self.request.user 通过当前用户过滤，或者实现其他更复杂的逻辑。与此同时，我们还可以把出版社对象添加到上下文中，供模板使用：

```
# ...

def get_context_data(self, **kwargs):
    # 先调用原来的实现，获取上下文
    context = super(PublisherBookList, self).get_context_data(**kwargs)

    # 添加出版社对象
    context['publisher'] = self.publisher
    return context ## 执行额外的操作
```

下面再介绍一个常见的需求：在调用通用视图前后做些额外工作。假设 Author 模型中有个 last_accessed 字段，用于记录这位作者的信息被人查看的最后时间：

```
# models.py
from django.db import models
```

```
class Author(models.Model):
    salutation = models.CharField(max_length=10)
    name = models.CharField(max_length=200)
    email = models.EmailField()
    headshot = models.ImageField(upload_to='author_headshots')
    last_accessed = models.DateTimeField()
```

当然，`DetailView` 类对这个字段一无所知，不过我们依然可以轻易自定义视图，及时更新这个字段。首先，我们要在 URL 配置中添加 `author-detail`，指向一个自定义的视图：

```
from django.conf.urls import url
from books.views import AuthorDetailView

urlpatterns = [
    #...
    url(r'^authors/(?P<pk>[0-9]+)/$', AuthorDetailView.as_view(),
        name='author-detail'),
]
```

然后要编写那个视图。`get_object` 是用于检索对象的方法，因此我们只需覆盖它，执行相关的调用：

```
from django.views.generic import DetailView
from django.utils import timezone
from books.models import Author

class AuthorDetailView(DetailView):

    queryset = Author.objects.all()

    def get_object(self):
        # 调用超类中的同名方法
        object = super(AuthorDetailView, self).get_object()

        # 记录最后访问日期
        object.last_accessed = timezone.now()
        object.save()
        # 返回对象
        return object
```

这里，URL 配置使用的分组名为 `pk`，这是 `DetailView` 过滤查询集合时查找主键所用的默认名称。

如果把这个分组命名为其他值，要在视图中设定 `pk_url_kwarg`。详情参见 `DetailView` 的文档。

10.6 接下来

本章只涵盖了 Django 自带的部分通用视图，不过基本思想几乎适用于所有通用视图。附录 C 将详细说明全部可用的通用视图，如果想充分利用通用视图的强大功能，建议你阅读。

本书对模型、模板和视图的高级用法的讨论至此结束。接下来的几章涵盖现代商业网站中十分常见的几个功能。首先，我们将探讨构建交互式网站的一个基本话题——用户管理。

第 11 章 在 Django 中验证用户的身份

现代的交互式网站有相当一部分比例有某种形式的用户交互，简单的交互有在博客中发表评论，全面的交互有新闻网站中编辑对文章的控制。如果网站以某种形式提供电商服务，验证付费顾客的身份并核准权限是基本需求。

就算只是管理用户（忘记用户名、密码，保持用户信息最新）也是一件痛苦的事。对程序员来说，编写一个身份验证系统更是难上加难。

幸运的是，Django 自带了一套系统，能管理用户的账户、分组和权限，并且实现了基于 cookie 的用户会话。

与 Django 的多数内置功能一样，这套系统的默认实现也完全可以扩展和定制，以满足具体项目的需求。下面就来探索这个系统。

11.1 概览

Django 的身份验证系统既能验证身份，也能核准权限。简单来说，身份验证是指确认用户是不是他声称的那个人，而权限核准是指确定通过身份验证的用户能做什么。这里，我们使用"身份验证"指代这两个任务。

Django 的身份验证系统包括：

- 用户
- 权限：二元（是或否）旗标，指明用户是否能执行特定的任务
- 分组：把标注和权限赋予多个用户的通用方式
- 可配置的密码哈希系统
- 管理身份验证和权限核准的表单
- 登录用户或限制内容的视图工具
- 可更换的后端系统

Django 的身份验证系统十分通用，没有提供 Web 身份验证系统中某些常用的功能。某些常用功能通过第三方包实现：

- 密码强度检查
- 登录尝试次数限制
- 通过第三方验证身份（如 OAuth）

11.2 使用 Django 的身份验证系统

Django 的身份验证系统默认实现了多数常见的需求，能处理相当多的任务，而且小心实现了密码和权限。如果你的项目不想使用默认的实现，Django 也允许对身份验证系统做深入地扩展和定制。

11.3 User 对象

User 对象是这个身份验证系统的核心，通常用于标识与网站交互的人，还用于限制访问、记录用户资料，以及把内容与创建人关联起来，等等。在 Django 的身份验证框架中，只有一个用户类存在，因此 superusers 或管理后台的 staff 用户只是设定了特殊属性的用户对象，而不是分属不同类的用户对象。默认用户主要有下面几个属性：

- username
- password
- email
- first_name
- last_name

11.3.1 创建超级用户

超级用户使用 createsuperuser 命令创建：

```
python manage.py createsuperuser --username=joe --email=joe@example.com
```

上述命令会提示你输入密码。输入密码后，立即创建指定的超级用户。如果没有指定 --username 或 --email 选项，会提示你输入这两个值。

11.3.2 创建用户

创建和管理用户最简单、最不易出错的方式是使用 Django 管理后台。Django 还内置了登录、退出和修改密码的视图和表单。本章后面会说明如何通过管理后台和普通的表单管理用户，现在先来看如何直接验证用户的身份。

创建用户最直接的方式是使用 create_user() 辅助函数：

```
>>> from django.contrib.auth.models import User
>>> user = User.objects.create_user('john', 'lennon@thebeatles.com',  'johnpassword')

# 此时，user 是一个 User 对象，而且已经保存到数据库中
# 如果想修改其他字段的值，可以继续修改属性
>>> user.last_name = 'Lennon'
>>> user.save()
```

11.3.3 修改密码

Django 不在用户模型中存储原始（明文）密码，只存储密码的哈希值。因此，不要试图直接处理用户的密码。正是因为这样，创建密码才要使用一个辅助函数。如果想修改用户的密码，有两个选择：

1. 在命令行中使用 manage.py changepassword username 命令修改用户的密码。这个命令会提示你输入两次密码。如果两次输入的内容匹配，立即修改密码。如果不指定用户名，这个命令会尝试修改与当前系统用户的用户名一致的那个用户的密码。
2. 还可以通过编程方式，使用 set_password() 方法修改：

```
>>> from django.contrib.auth.models import User
>>> u = User.objects.get(username='john')
```

```
>>> u.set_password('new password')
>>> u.save()
```

如果启用了 SessionAuthenticationMiddleware，修改用户的密码后，那个用户的所有会话都会退出。

11.4 权限和权限核准

Django 自带了一个简单的权限系统。通过它可以为指定的用户和用户组赋予权限。Django 管理后台就用到了这个系统，当然也欢迎在你自己的代码中使用。Django 管理后台使用权限控制下述操作：

- 限制有某种对象的"添加"权限才能查看"添加"表单和添加对象。
- 限制有某种对象的"修改"权限才能查看修改列表、"修改"表单和修改对象。
- 限制有某种对象的"删除"权限才能删除对象。

权限不仅可以在一种对象上设定，也可以在具体的对象实例上设定。使用 ModelAdmin 类提供的 has_add_permission()、has_change_permission() 和 has_delete_permission() 方法可以定制同一类型不同实例的权限。User 对象有两个多对多字段，分别是 groups 和 user_permissions。User 对象访问相关对象的方式与其他 Django 模型一样。

11.4.1 默认权限

在 INSTALLED_APPS 设置中列出 django.contrib.auth 后，安装的各个应用中的每个 Django 模型默认都有三个权限：添加、修改和删除。每次运行 manage.py migrate 命令创建新模型时都会为其赋予这三个权限。

11.4.2 分组

django.contrib.auth.models.Group 模型是为用户分类的通用方式，这样便可以为一批用户赋予权限或添加其他标注。用户所属的分组数量不限。一个分组中的用户自动获得赋予那个分组的权限。例如"Site editors"分组有 can_edit_home_page 权限，那么其中的任何一个用户都有这个权限。

除了权限之外，分组还是为用户分类的便捷方式，分组后可以给用户添加标签，或者扩展功能。例如，可以创建"Special users"分组，然后编写代码，允许这一组中的用户访问只有会员才能查看的内容，或者发送只给会员看的电子邮件。

11.4.3 通过编程方式创建权限

除了可以在模型的 Meta 类中定制权限之外，还可以直接创建权限。例如，为 books 应用中的 BookReview 模型赋予 can_publish 权限：

```
from books.models import BookReview
from django.contrib.auth.models import Group, Permission
from django.contrib.contenttypes.models import ContentType

content_type = ContentType.objects.get_for_model(BookReview)
permission = Permission.objects.create(codename='can_publish',
                                       name='Can Publish Reviews',
                                       content_type=content_type)
```

然后，可以通过 User 的 user_permissions 属性把这个权限赋予一个用户，或者通过 Group 的 permissions 属性把这个权限赋予一个分组。

11.4.4 权限缓存

首次检查用户对象的权限时，ModelBackend 会缓存权限。这在"请求-响应"循环中是没问题的，因为添加权限后往往不会立即检查（例如在管理后台中）。

如果添加权限后需要立即检查，例如在测试或视图中，最简单的解决方法是重新从数据库中获取用户对象。例如：

```
from django.contrib.auth.models import Permission, User
from django.shortcuts import get_object_or_404

def user_gains_perms(request, user_id):
    user = get_object_or_404(User, pk=user_id)
    # 只要检查了权限，就会把当前的权限缓存起来
    user.has_perm('books.change_bar')

    permission = Permission.objects.get(codename='change_bar')
    user.user_permissions.add(permission)

    # 检查的是缓存的权限
    user.has_perm('books.change_bar')  # False

    # 重新请求 User 实例
    user = get_object_or_404(User, pk=user_id)

    # 重新缓存权限
    user.has_perm('books.change_bar')  # True

    # ...
```

11.5 在 Web 请求中验证身份

Django 使用会话和中间件把身份验证系统插入 request 对象，为每个请求提供 request.user 属性，表示当前用户。如果未登陆，这个属性的值是一个 AnonymousUser 实例，否则是是一个 User 实例。这两种情况可以使用 is_authenticated() 方法区分，例如：

```
if request.user.is_authenticated():
    # 处理通过身份验证的用户
else:
    # 处理匿名用户
```

11.5.1 如何登录用户

在视图中使用 login() 登录用户。它的参数是一个 HttpRequest 对象和一个 User 对象。login() 使用 Django 的会话框架把用户的 ID 保存到会话中。注意，匿名期间设定的会话数据在用户登录后依然存在。下述示例展示 authenticate() 和 login() 的用法：

```
from django.contrib.auth import authenticate, login

def my_view(request):
    username = request.POST['username']
    password = request.POST['password']
```

```
user = authenticate(username=username, password=password)
if user is not None:
    if user.is_active:
        login(request, user)
        # 重定向到成功登录页面
    else:
        # 返回“账户未激活”错误消息
else:
    # 返回“无效登录”错误消息
```

先调用 authenticate()

自己动手登录用户时，必须在 login() 之前调用 authenticate()。authenticate() 在 User 对象上设定一个属性，指明成功验证用户身份的是哪个身份验证后端，而登录过程中需要使用这个信息。如果直接登录从数据库中检索的用户对象，Django 报错。

11.5.2 如何退出用户

在视图中退出通过 login() 登录的用户使用 logout()。这个函数的参数是一个 HttpRequest 对象，而且没有返回值。例如：

```
from django.contrib.auth import logout

def logout_view(request):
    logout(request)
    # 重定向到成功退出页面
```

注意，如果用户未登录，logout() 函数不报错。调用 logout() 函数后，当前请求的会话数据完全清除，所有数据将被删除。这样能避免其他人在登录的 Web 浏览器中访问用户之前的会话数据。

如果想让会话中的数据在退出后依然可用，调用 logout() 函数之后再把数据存入会话。

11.5.3 限制已登录用户的访问

直接方式

限制访问页面简单直接的方式是检查 request.user.is_authenticated()，如果未通过，可以重定向到登录页面：

```
from django.shortcuts import redirect

def my_view(request):
    if not request.user.is_authenticated():
        return redirect('/login/?next=%s' % request.path)
    # ...
```

也可以显示一个错误消息：

```
from django.shortcuts import render

def my_view(request):
    if not request.user.is_authenticated():
```

```
        return render(request, 'books/login_error.html')
    # ...
```

login_required 装饰器

若想再简单一些，可以使用便利的 login_required() 装饰器：

```
from django.contrib.auth.decorators import login_required

@login_required
def my_view(request):
    ...
```

login_required() 的作用如下：

- 如果用户未登录，重定向到 LOGIN_URL，并把当前绝对路径添加到查询字符串中。例如：/accounts/login/?next=/reviews/3/。
- 如果用户已登录，正常执行视图。视图代码可以放心假定用户已登录。

默认，成功通过身份验证后重定向的目标路径存储在名为 next 的查询字符串参数中。如果想为这个参数提供其他名称，可以设定 login_required() 可选的 redirect_field_name 参数：

```
from django.contrib.auth.decorators import login_required

@login_required(redirect_field_name='my_redirect_field')
def my_view(request):
    ...
```

注意，如果为 redirect_field_name 提供了值，可能还要定制登录模板，因为模板上下文中存储重定向路径的变量名是 redirect_field_name 的值，而不再是默认的 next。login_required() 还有个可选的 login_url 参数。例如：

```
from django.contrib.auth.decorators import login_required

@login_required(login_url='/accounts/login/')
def my_view(request):
    ...
```

注意，如果不指定 login_url 参数，要确保把 LOGIN_URL 设为正确的登录视图。例如，使用默认配置时，要把下述代码添加到 URL 配置中：

```
from django.contrib.auth import views as auth_views

url(r'^accounts/login/$', auth_views.login),
```

LOGIN_URL 的值还可以是视图函数名称或具名 URL 模式。这样，无需修改设置就可以在 URL 配置中自由映射登录视图。

> **注意**
>
> login_required 装饰器不检查用户的 is_active 旗标。

根据测试条件限制访问

如果想根据权限或其他测试条件限制访问，要做的基本上与前一节所述的一样。简单的方式是在视图中直接测试 request.user。例如，下述示例检查用户的电子邮件是否由指定域名提供：

```
def my_view(request):
    if not request.user.email.endswith('@example.com'):
        return HttpResponse("You can't leave a review for this book.")
    # ...
```

更简单的方式是使用便利的 user_passes_test 装饰器：

```
from django.contrib.auth.decorators import user_passes_test

def email_check(user):
    return user.email.endswith('@example.com')

@user_passes_test(email_check)
def my_view(request):
    ...
```

user_passes_test() 有个必须指定的参数：一个可调用对象，其参数为一个 User 对象，允许用户查看页面时返回 True。注意，user_passes_test() 不自动检查用户是不是匿名的。这个装饰器有两个可选的参数：

1. login_url。指定一个 URL，把未通过测试的用户重定向到那里。可以设为登录页面，如果不指定则使用 LOGIN_URL 的值。
2. redirect_field_name。与 login_required() 中的作用一样。设为 None 时，URL 中没有这个查询字符串参数。如果把用户重定向到登录页面之外的页面，那就没有“下一个页面”，因此无需这个参数。

例如：

```
@user_passes_test(email_check, login_url='/login/')
def my_view(request):
    ...
```

permission_required 装饰器

检查用户有没有特定权限是比较常见的任务。鉴于此，Django 提供了一种简便的方式——permission_required() 装饰器：

```
from django.contrib.auth.decorators import permission_required

@permission_required('reviews.can_vote')
def my_view(request):
    ...
```

与 has_perm() 方法一样，参数名称的形式为“<app label>.<permission codename>”（例如，reviews.can_vote 是 reviews 应用中某个模型定义的权限）。这个装饰器的参数也可以是一个权限列表。注意，permission_required() 也有可选的 login_url 参数。例如：

```
from django.contrib.auth.decorators import permission_required

@permission_required('reviews.can_vote', login_url='/loginpage/')
def my_view(request):
```

```
    ...
```

与 login_required() 装饰器一样，login_url 的默认值是 LOGIN_URL。如果指定了 raise_exception 参数，这个装饰器不会重定向到登录页面，而是抛出 PermissionDenied 异常，显示 403（HTTP Forbidden）视图。

修改密码后作废会话

如果 AUTH_USER_MODEL 继承自 AbstractBaseUser，或者定制了 get_session_auth_hash() 方法，通过身份验证的会话包含这个方法返回的哈希值。对 AbstractBaseUser 来说，返回的是密码字段的 Hash Message Authentication Code（HMAC）。

如果启用了 SessionAuthenticationMiddleware，Django 会验证随各个请求发送的哈希值是否与服务器端计算的匹配。这样，修改密码后，用户的所有会话都会失效，从而退出。

Django 自带的密码修改视图，django.contrib.auth.views.password_change() 和 django.contrib.auth 中的 user_change_password，在用户修改自己的密码后会使用新的密码哈希值更新会话，因此不会退出。如果使用自定义的密码修改视图，想具有类似的行为，使用这个函数：

```
django.contrib.auth.decorators.update_session_auth_hash(request, user)
```

这个函数的参数是当前请求对象和更新后的用户对象，从后一个参数中获取新哈希值后更新会话。示例用法如下：

```
from django.contrib.auth import update_session_auth_hash

def password_change(request):
    if request.method == 'POST':
        form = PasswordChangeForm(user=request.user, data=request.POST)
        if form.is_valid():
            form.save()
            update_session_auth_hash(request, form.user)
    else:
        ...
```

get_session_auth_hash() 要使用 SECRET_KEY，因此更新网站的密钥后，现有会话都会作废。

11.6 身份验证视图

Django 为登录、退出和密码管理提供了视图。这些视图使用 auth 包中内置的表单，不过也可以传入自己编写的视图。Django 没有为身份验证视图提供默认的模板，不过下文将说明各个视图的模板上下文。

在项目中使用这些视图要实现不同的方法，不过最简单也是最常见的做法是把 django.contrib.auth.urls 提供的 URL 配置添加到项目的 URL 配置中。例如：

```
urlpatterns = [url('^', include('django.contrib.auth.urls'))]
```

这样，各个视图在默认的 URL 上（后文详述）。

这些内置的视图都返回一个 TemplateResponse 实例，这样便于在渲染之前定制响应数据。多数内置的身份验证视图提供了 URL 名称，易于引用。

11.6.1 login 视图

登录用户。

默认 URL：/login/。

可选参数：

- template_name：这个视图使用的模板名称。默认为 registration/login.html。
- redirect_field_name：GET 参数中指定登录后重定向 URL 的字段名称。默认为 next。
- authentication_form：验证身份的可调用对象（通常是一个表单类）。默认为 AuthenticationForm。
- current_app：一个提示，指明当前视图所在的应用。详情参见 7.8.1 节。
- extra_context：一个字典，包含额外的上下文数据，随默认的上下文数据一起传给模板。

login 视图的作用如下：

- 如果通过 GET 调用，显示登录表单，其目标地址与当前 URL 一样。稍后详述。
- 如果通过 POST 调用，发送用户提交的凭据，尝试登录用户。如果登录成功，重定向到 next 指定的 URL。如果没有 next，重定向到 LOGIN_REDIRECT_URL（默认为 /accounts/profile/）。如果登录失败，重新显示登录表单。

登录视图的模板由你提供，模板文件默认名为 registration/login.html。

模板上下文：

- form：表示 AuthenticationForm 的 Form 对象。
- next：成功登录后重定向的目标 URL。自身可能也包含查询字符串。
- site：当前 Site，根据 SITE_ID 设置确定。如果未安装网站框架，其值默认为 RequestSite 实例，从当前 HttpRequest 对象中获取网站名称和域名。
- site_name：site.name 的别名。如果未安装网站框架，其值为 request.META['SERVER_NAME'] 的值。

如果不想把模板命名为 registration/login.html，可以为 URL 配置提供额外的参数，设定 template_name 参数。

11.6.2 logout 视图

退出用户。

默认 URL：/logout/

可选的参数：

- next_page：退出后重定向的目标 URL。
- template_name：一个模板全名，在用户退出后显示。如果未提供这个参数，默认为 registration/logged_out.html。
- redirect_field_name：GET 参数中指定退出后重定向 URL 的字段名称。默认为 next。如果提供这个参数，next_page 将被覆盖。
- current_app：一个提示，指明当前视图所在的应用。详情参见 7.8.1 节。

- extra_context：一个字典，包含额外的上下文数据，随默认的上下文数据一起传给模板。

模板上下文：

- title：本地化之后的字符串“Logged out”。
- site：当前 Site，根据 SITE_ID 设置确定。如果未安装网站框架，其值默认为 RequestSite 实例，从当前 HttpRequest 对象中获取网站名称和域名。
- site_name：site.name 的别名。如果未安装网站框架，其值为 request.META['SERVER_NAME'] 的值。

11.6.3 logout_then_login 视图

退出用户，然后重定向到登录页面。

默认 URL：未提供。

可选的参数：

- login_url：重定向到的登录页面的 URL。如果未提供，默认为 LOGIN_URL。
- current_app：一个提示，指明当前视图所在的应用。详情参见 7.8.1 节。
- extra_context：一个字典，包含额外的上下文数据，随默认的上下文数据一起传给模板。

11.6.4 password_change 视图

让用户修改密码。

默认 URL：/password_change/

可选的参数：

- template_name：完整的模板名称，用于显示密码修改表单。如果未提供，默认为 registration/password_change_form.html。
- post_change_redirect：成功修改密码后重定向的目标 URL。
- password_change_form：自定义的修改密码表单，必须接受 user 关键字参数。这个表单负责修改用户的密码。默认为 PasswordChangeForm。
- current_app：一个提示，指明当前视图所在的应用。详情参见 7.8.1 节。
- extra_context：一个字典，包含额外的上下文数据，随默认的上下文数据一起传给模板。

模板上下文：

- form：密码修改表单（参见前面的 password_change_form）。

11.6.5 password_change_done 视图

这个页面在用户修改密码后显示。

默认 URL：/password_change_done/

可选的参数：

- template_name：要使用的模板全名。如果未提供，默认为 registration/password_change_done.html。
- current_app：一个提示，指明当前视图所在的应用。详情参见 7.8.1 节。
- extra_context：一个字典，包含额外的上下文数据，随默认的上下文数据一起传给模板。

11.6.6 password_reset 视图

生成一次性链接，发给用户注册时填写的电子邮件地址，让用户重设密码。

如果系统中没有用户提供的电子邮件地址，这个视图不发送电子邮件，而且用户也不会看到错误消息。这样能防止泄露信息，防止被潜在的攻击者利用。如果想为这种情况提供错误消息，可以定义 PasswordResetForm 的子类，把它赋值给 password_reset_form 参数。

密码被标记为不可用的用户不允许请求重设密码，以防使用外部身份验证源（如 LDAP）时误用。注意，此时用户看不到错误消息，也不会发送邮件，这是为了防止暴漏用户账户。

默认 URL：/password_reset/

可选参数：

- template_name：完整的模板名称，用于显示密码重设表单。如果未提供，默认为 registration/password_reset_form.html/
- email_template_name：完整的模板名称，用于生成带有密码重设链接的电子邮件。如果未提供，默认为 registration/password_reset_email.html。
- subject_template_name：完整的模板名称，用于生成密码重设邮件的主题。如果未提供，默认使用 registration/password_reset_subject.txt。
- password_reset_form：用于获取请求重设的用户的电子邮件。默认为 PasswordResetForm。
- token_generator：检查一次性链接的类的实例。默认为 default_token_generator，它是 django.contrib.auth.tokens.PasswordResetTokenGenerator 的实例。
- post_reset_redirect：成功请求重设密码之后重定向的目标 URL。
- from_email：一个有效的电子邮件地址。Django 默认使用 DEFAULT_FROM_EMAIL。
- current_app：一个提示，指明当前视图所在的应用。详情参见 7.8.1 节。
- extra_context：一个字典，包含额外的上下文数据，随默认的上下文数据一起传给模板。
- html_email_template_name：完整的模板名称，用于生成内容类型为 text/html 的多部分（multipart）电子邮件。默认不发送 HTML 格式的电子邮件。

模板上下文：

- form：用于重设用户密码的表单（参见上面的 password_reset_form）。

电子邮件模板上下文：

- email：user.email 的别名。
- user：根据 email 表单字段确认的当前用户。只有已激活用户（即 User.is_active 的值为 True）才能重设密码。
- site_name：site.name 的别名。如果未安装网站框架，其值为 request.META['SERVER_NAME'] 的值。
- domain：site.domain 的别名。如果未安装网站框架，其值为 request.get_host() 的值。

- protocol：http 或 https。
- uid：base64 编码的用户主键。
- token：检查重设链接是否有效的令牌。

registration/password_reset_email.html 示例（电子邮件正文模板）：

```
Someone asked for password reset for email {{ email }}. Follow the link below:
{{ protocol}}://{{ domain }}{% url 'password_reset_confirm' uidb64=uid token=token %}
```

邮件主题模板也使用上述模板上下文。主题必须是单行纯文本字符串。

11.6.7 password_reset_done 视图

成功把密码重设链接发送给用户后显示的页面。如果 password_reset() 视图没有设定 post_reset_redirect URL，默认使用这个视图。

默认 URL：/password_reset_done/

> **提示**
>
> 如果系统中没有用户提供的电子邮件地址、用户未激活或密码不可用，也会把用户重定向到这个页面，但是不发送电子邮件。

可选参数：

- template_name：要使用的模板完整名称。如果未提供，默认为 registration/password_reset_done.html。
- current_app：一个提示，指明当前视图所在的应用。详情参见 7.8.1 节。
- extra_context：一个字典，包含额外的上下文数据，随默认的上下文数据一起传给模板。

11.6.8 password_reset_confirm 视图

呈现输入新密码的表单。

默认 URL：/password_reset_confirm/

可选参数：

- uidb64：base64 编码的用户 ID。默认为 None。
- token：检查密码是否有效的令牌。默认为 None。
- template_name：模板的完整名称，显示密码确认视图。默认为 registration/password_reset_confirm.html。
- token_generator：检查密码的类的实例。默认为 default_token_generator，它是 django.contrib.auth.tokens.PasswordResetTokenGenerator 的实例。
- set_password_form：用于设定密码的表单。默认为 SetPasswordForm。
- post_reset_redirect：重设密码后重定向的目标 URL。默认为 None。
- current_app：一个提示，指明当前视图所在的应用。详情参见 7.8.1 节。

- extra_context：一个字典，包含额外的上下文数据，随默认的上下文数据一起传给模板。

模板上下文：

- form：设定新密码的表单（参见上面的 set_password_form）。
- validlink：布尔值，链接有效（uidb64 和 token 都检查）或尚未使用时为 True。

11.6.9 password_reset_complete 视图

呈现一个视图，告诉用户成功修改了密码。

默认 URL：/password_reset_complete/

可选参数：

- template_name：显示这个视图的模板完整名称。默认为 registration/password_reset_complete.html。
- current_app：一个提示，指明当前视图所在的应用。详情参见 7.8.1 节。
- extra_context：一个字典，包含额外的上下文数据，随默认的上下文数据一起传给模板。

11.6.10 redirect_to_login 辅助函数

为了便于在视图中实现所需的访问限制，Django 提供了 redirect_to_login 辅助函数。它的作用是重定向到登录页面，成功登录后再返回之前请求的 URL。

必要参数：

- next：成功登录后重定向的目标 URL。

可选参数：

- login_url：重定向的登录页面的 URL。如果未提供，默认为 LOGIN_URL。
- redirect_field_name：指定登录后重定向的目标 URL 的 GET 字段名称。如果设定，覆盖 next。

11.6.11 内置的表单

如果不想使用内置的视图，也不想为这些功能编写表单，可以使用身份验证系统在 django.contrib.auth.forms 中内置的几个表单（表 11-1）。

这些内置的表单对用户模型有些假设，如果自定义了用户模型，可能要自己动手为身份验证系统编写表单。

表 11-1：Django 内置的身份验证表单

表单名	说明
AdminPasswordChangeForm	在管理后台中用于修改用户密码的表单。第一个位置参数是用户对象。
AuthenticationForm	登录表单。请求对象是第一个位置参数，存储在表单中，供子类使用。
PasswordChangeForm	修改密码的表单。
PasswordResetForm	用于生成并发送带有重设密码链接的电子邮件。

（续）

表单名	说明
SetPasswordForm	让用户修改密码的表单，无需输入旧密码。
UserChangeForm	在管理后台中用于修改用户信息和权限的表单。
UserCreationForm	创建新用户的表单。

11.7 模板中的身份验证数据

使用 RequestContext 时，当前登录用户及其权限可通过模板上下文访问。

11.7.1 用户

渲染模板的 RequestContext 时，当前登录用户，不管是 User 实例还是 AnonymousUser 实例，都存储在模板变量 {{ user }} 中：

```
{% if user.is_authenticated %}
    <p>Welcome, {{ user.username }}. Thanks for logging in.</p>
{% else %}
    <p>Welcome, new user. Please log in.</p>
{% endif %}
```

如果使用的不是 RequestContext，这个模板上下文变量不可用。

11.7.2 权限

当前登录用户的权限存储在模板变量 {{ perms }} 中。它的值是 django.contrib.auth.context_processors.PermWrapper 的一个实例，对模板友好。在 {{ perms }} 对象中，单属性查找由 User.has_module_perms 代理。只要当前登录用户在 foo 应用中有权限，下述示例就返回 True：

```
{{ perms.foo }}
```

两层属性查找由 User.has_perm 代理。如果当前登录用户有 foo.can_vote 权限，下述示例返回 True：

```
{{ perms.foo.can_vote }}
```

因此，在模板中可以使用 {% if %} 语句检查权限：

```
{% if perms.foo %}
    <p>You have permission to do something in the foo app.</p>
    {% if perms.foo.can_vote %}
        <p>You can vote!</p>
    {% endif %}
    {% if perms.foo.can_drive %}
        <p>You can drive!</p>
    {% endif %}
{% else %}
    <p>You don't have permission to do anything in the foo app.</p>
{% endif %}
```

此外，还可以使用 {% if in %} 语句检查权限。例如：

```
{% if 'foo' in perms %}
    {% if 'foo.can_vote' in perms %}
        <p>In lookup works, too.</p>
    {% endif %}
{% endif %}
```

11.8 在管理后台中管理用户

django.contrib.admin 和 django.contrib.auth 都安装时，在管理后台可以方便地查看和管理用户、分组和权限。用户可以像其他 Django 模型那样创建和删除。此外，还可以创建分组，并把权限赋予用户或分组。管理后台还存储着用户编辑日志，可供查看。

11.8.1 创建用户

在管理后台的首页，“Authentication and Authorization”部分中有个“Users”链接。点击后看到的是用户管理界面（图 11-1）。

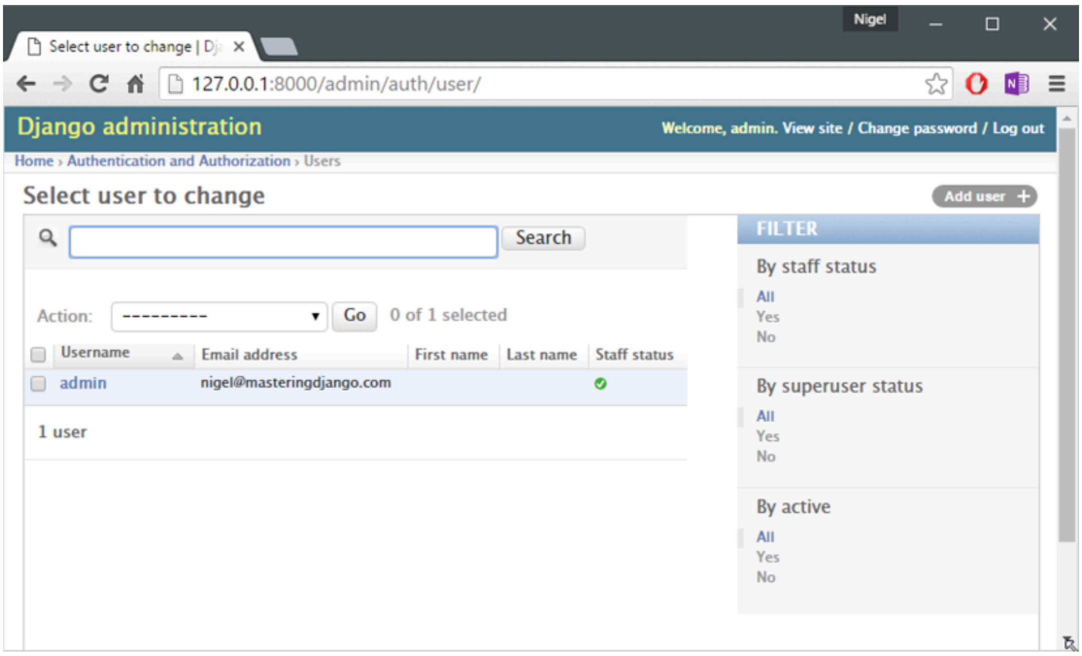


图 11-1：Django 管理后台中的用户管理界面

“Add user”页面与管理后台标准的页面不同，要先填写用户名和密码才能编辑其余的字段（图 11-2）。

> **提示**
>
> 如果想让用户在 Django 管理后台创建用户，要赋予他添加和修改用户的权限（即“Add user”和“Change user”权限）。如果用户只有权添加用户而无权修改用户，他就无法添加用户。为什么？因为有权添加用户就能创建超级用户，如此以来就能修改其他用户。所以，出于安全考虑，Django 强制要求必须兼具“添加”和“修改”两个权限。

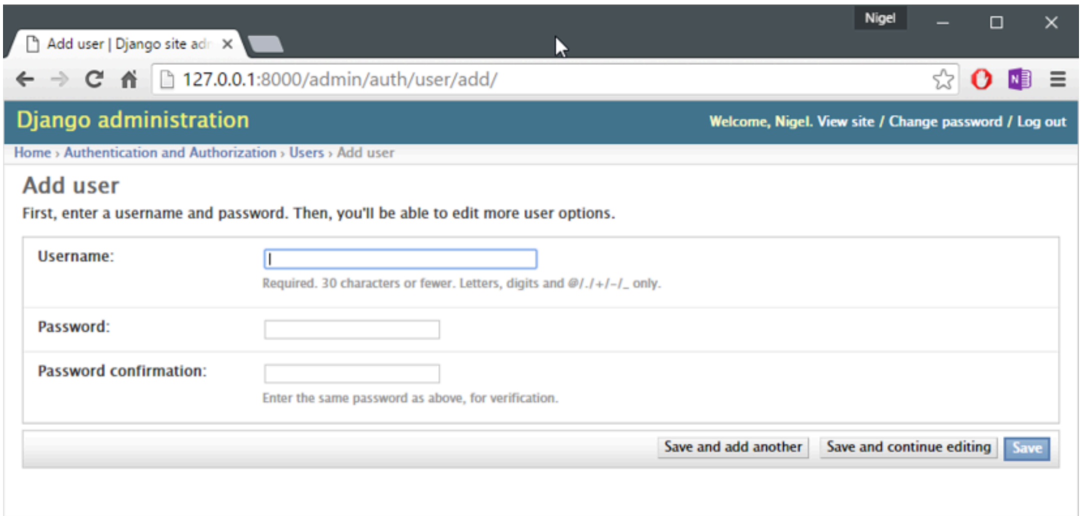


图 11-2：Django 管理后台的添加用户界面

11.8.2 修改密码

管理后台不显示用户的密码（数据库中也不存储），但是显示着存储的密码密文。此外，还有指向密码修改表单的链接，让管理员修改用户的密码（图 11-3）。

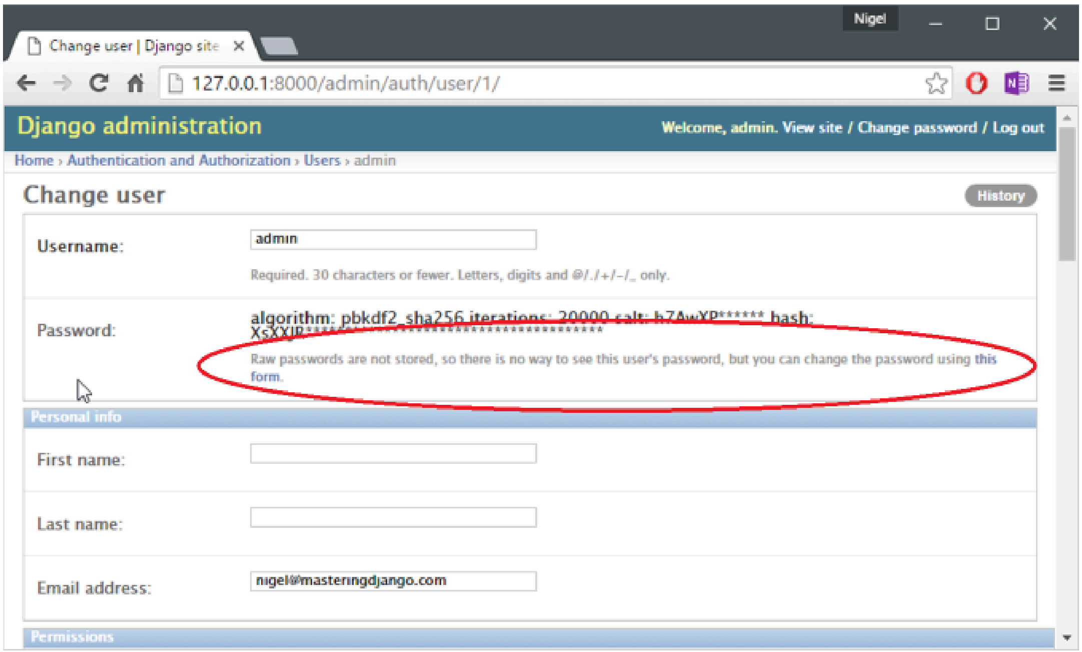


图 11-3：修改密码的链接（圈出部分）

点击那个链接后，会显示修改密码表单（图 11-4）。

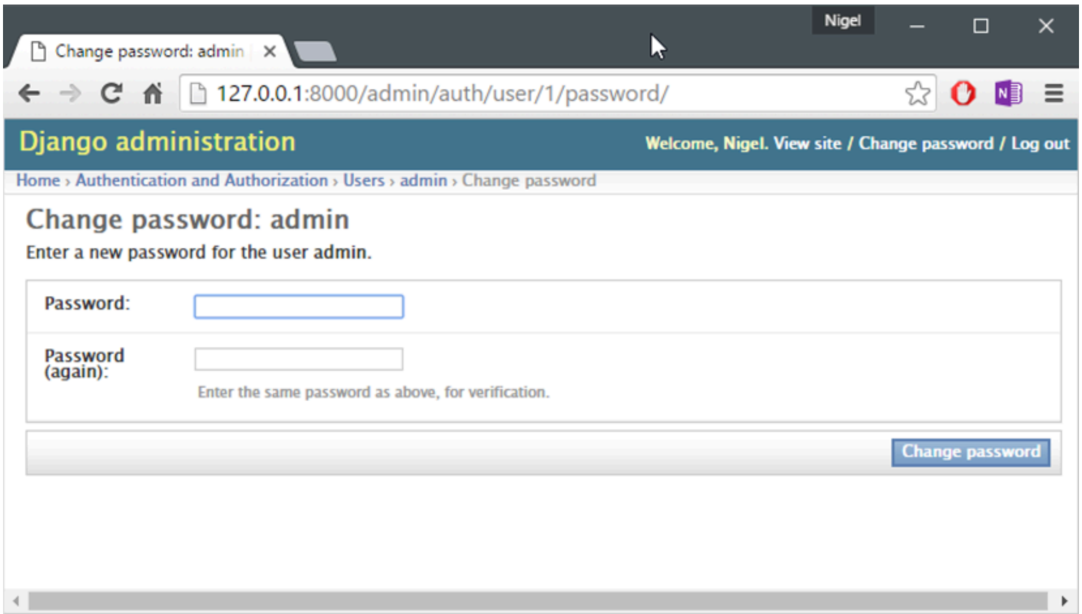


图 11-4：Django 管理后台的密码修改表单

11.9 密码管理

如非必要，不要重新实现密码管理功能，Django 提供的功能足够安全和灵活。本节说明 Django 如何存储密码、如何配置哈希方式，以及处理哈希密码的实用工具。

11.9.1 Django 如何存储密码

Django 提供了一个灵活的密码存储系统，默认使用 PBKDF2 算法。`User` 对象的 `password` 属性是这种格式的字符串：

```
<algorithm>$<iterations>$<salt>$<hash>
```

这是存储用户密码的各个部分，之间以美元符号分隔。各部分分别是：哈希算法、算法的迭代次数（工作系数）、随机盐值和最终得到的密码哈希值。

`algorithm` 是 Django 支持的某种单向哈希（或称“密码存储”）算法（参见下文）。`iterations` 是在哈希值上运用算法的次数。`salt` 是使用的随机种子，而 `hash` 是单向算法得到的结果。Django 默认使用 `PBKDF2` 算法，得到 SHA256 哈希值。这是 NIST 推荐使用的密码增强机制，对多数用户来说足够了。这种算法相当安全，需要用大量时间计算才能破解。然而，根据具体的需求，你可能想选择其他的算法，甚至自己动手实现一种算法，满足特殊的安全条件。再次说明，多数用户无需这么做。如果你不确定自己该不该这么做，那有可能就不需要。

如果确实需要这么做，请接着读：Django 根据 `PASSWORD_HASHERS` 设置选择算法。这个设置的值是一个类列表，列出你所安装的 Django 支持的哈希算法。其中第一个元素（即 `settings.PASSWORD_HASHERS[0]`）用于存储密码，其余的都可用于检查现有密码。

也就是说，如果想使用其他算法，要修改 `PASSWORD_HASHERS` 设置，把你选择的算法列在第一位。`PASSWORD_HASHERS` 的默认值如下：

```
PASSWORD_HASHERS = [
    'django.contrib.auth.hashers.PBKDF2PasswordHasher',
```

```
        'django.contrib.auth.hashers.PBKDF2SHA1PasswordHasher',
        'django.contrib.auth.hashers.BCryptSHA256PasswordHasher',
        'django.contrib.auth.hashers.BCryptPasswordHasher',
        'django.contrib.auth.hashers.SHA1PasswordHasher',
        'django.contrib.auth.hashers.MD5PasswordHasher',
        'django.contrib.auth.hashers.CryptPasswordHasher',
    ]
```

因此，Django 使用 PBKDF2 算法存储所有密码，但是支持使用 PBKDF2SHA1、bcrypt、SHA1 等算法检查存储的密码。接下来的几节说明修改这一设置的常见方式。

11.9.2 让 Django 使用 bcrypt

bcrypt 是一种流行的密码存储算法，专门针对长期存储的密码。Django 默认没有使用这个算法，因为需要第三方库。不过，因为有很多人想使用，所以 Django 为 bcrypt 提供了最低的支持。

若想把 bcrypt 设为默认的存储算法，这么做：

1. 安装 `bcrypt` 库。可以执行 `pip install django[bcrypt]` 命令，或者下载之后执行 `python setup.py install` 命令安装。
2. 修改 `PASSWORD_HASHERS` 设置，把 `BCryptSHA256PasswordHasher` 列在第一位。即，把下述代码放到设置文件中：

```
PASSWORD_HASHERS = [
    'django.contrib.auth.hashers.BCryptSHA256PasswordHasher',
    'django.contrib.auth.hashers.BCryptPasswordHasher',
    'django.contrib.auth.hashers.PBKDF2PasswordHasher',
    'django.contrib.auth.hashers.PBKDF2SHA1PasswordHasher',
    'django.contrib.auth.hashers.SHA1PasswordHasher',
    'django.contrib.auth.hashers.MD5PasswordHasher',
    'django.contrib.auth.hashers.CryptPasswordHasher',
]
```

（其他元素仍要放在这个列表中，否则 Django 不会更新密码。参见下文。）

搞定！现在 Django 将使用 bcrypt 为默认的存储算法。

BCryptPasswordHasher 会截断密码

按设计，bcrypt 在第 72 个字符处截断密码，这意味着 `bcrypt(password_with_100_chars) == bcrypt(password_with_100_chars[:72])`。`BCryptPasswordHasher` 没有做特殊处理，因此也遵守这个隐秘的密码长度限制。`BCryptSHA256PasswordHasher` 打破了这一限制，它首先使用 sha256 计算密码的哈希值。这样能避免密码被截断，因此是首选。截断的实际后果不太严重，因为多数用户的密码不会超过 72 个字符，即便被截断了，暴力破解 bcrypt 所需的时间也是天文数字。尽管如此，“小心驶得万年船”，我们还是推荐使用 `BCryptSHA256PasswordHasher`。

11.9.3 其他 bcrypt 实现

在 Django 中使用 bcrypt 还有其他方式，但是 Django 对 bcrypt 的支持不直接支持那些方式。为了升级密码，需要把数据库中的哈希值修改成 `bcrypt$(raw bcrypt output)` 格式。

11.9.4 增加工作系数

PBKDF2 和 bcrypt 算法会对哈希值多次迭代，这么做是为了拖延攻击者，让破解哈希密码的难度更大。然而，随着计算机能力的增强，迭代的次数应随之增加。

Django 开发团队选择了一个合理的默认值（每次发布 Django 新版都会增加），不过你可能想根据自己的安全需求和可用的处理能力调大或调小。为此，要定义相应算法的子类，然后覆盖 `iterations` 属性。

例如，增加默认的 PBKDF2 算法的迭代次数的步骤如下：

1. 定义 `django.contrib.auth.hashers.PBKDF2PasswordHasher` 的子类：

```
from django.contrib.auth.hashers import PBKDF2PasswordHasher

class MyPBKDF2PasswordHasher(PBKDF2PasswordHasher):
    iterations = PBKDF2PasswordHasher.iterations * 100
```

2. 把这个类保存在项目中的某个位置。例如，可以放在 `myproject/hashers.py` 文件中。
3. 把自定义的哈希类设为 `PASSWORD_HASHERS` 中的第一个元素：

```
PASSWORD_HASHERS = [
    'myproject.hashers.MyPBKDF2PasswordHasher',
    'django.contrib.auth.hashers.PBKDF2PasswordHasher',
    # ... #
]
```

结束！现在，你的 Django 在使用 PBKDF2 算法时将迭代更多次。

11.9.5 密码升级

用户登录时，如果密码不是以选中的算法存储的，Django 会自动使用新算法升级密码。这意味着，旧的 Django 应用程序能在用户登录时变得更安全，也意味着你可以随时切换为新推出的（和更好的）存储算法。

然而，Django 只会对 `PASSWORD_HASHERS` 中列出的算法进行升级，因此，换用新算法时一定不能把旧的算法从列表中删除。如若不然，使用列表中没有的算法存储的密码将无法升级。修改 PBKDF2 算法的迭代次数后也会升级密码。

11.9.6 手动管理用户的密码

`django.contrib.auth.hashers` 模块提供了一系列用于创建和验证哈希密码的函数。这些函数可以独立于 `User` 模型使用。

如果想自己动手比较纯文本密码和数据库中的密码哈希值，使用 `check_password()` 函数。它有两个参数：要检查的纯文本密码和数据库中 `password` 字段的值。如果二者匹配，返回 `True`，否则返回 `False`。

`make_password()` 根据应用程序使用的算法创建哈希密码。它有一个必要的参数，即纯文本密码。

如果不想使用默认值（`PASSWORD_HASHERS` 设置中的第一个元素），还可以提供一个盐值和哈希算法。目前支持的算法有：`'pbkdf2_sha256'`、`'pbkdf2_sha1'`、`'bcrypt_sha256'`、`'bcrypt'`、`'sha1'`、`'md5'`、`'unsalted_md5'`（只是为了向后兼容）和 `'crypt'`（要安装 `crypt` 库）。

如果密码参数的值是 `None`，返回一个不可用的密码（绝对通不过 `check_password()` 的检查）。

is_password_usable() 检查指定字符串是不是可能通过 check_password() 验证的哈希密码。

11.10 自定义身份验证

多数情况下，Django 自带的身份验证系统足够用了，但是开箱即用的默认系统可能无法满足你的需求。若想定制，你要了解自带系统的哪些部分是可扩展的或可替换的。

身份验证后端提供了一个可扩展的系统，以防你想使用别的服务验证 User 模型存储的用户名和密码。你可以为模型赋予其他权限，然后使用 Django 的权限核准系统检测。你可以扩展默认的 User 模型，或者完全替换成自定义的模型。

11.10.1 其他身份验证源

有时，可能需要连接其他身份验证源，即用户名、密码和身份验证方法来自别处。

例如，你的公司可能已经搭建好了 LDAP，用于存储每个员工的用户名和密码。如果分别为 LDAP 和 Django 开发的应用程序提供一套账户，对网络管理员和用户自身都是件麻烦事。

考虑到这种情况，Django 身份验证系统支持连接其他身份验证源。你可以覆盖 Django 默认基于数据库的模式，或者把默认系统与其他系统连接起来。

11.10.2 指定身份验证后端

Django 在背后维护着一个用于验证身份的后端列表。调用 authenticate() 时（参见 11.5.1 节），Django 在所有身份验证后端中一一尝试。如果第一个后端验证失败，再试第二个，然后一直下去，直到试完所有后端为止。

身份验证后端列表由 AUTHENTICATION_BACKENDS 设置指定。它的值是一个 Python 路径名列表，指向知道如何验证身份的 Python 类。这些类可以在 Python 路径中的任何位置。AUTHENTICATION_BACKENDS 的默认值为：

```
['django.contrib.auth.backends.ModelBackend']
```

这是一个基本的身份验证后端，检查 Django 用户数据库，并查询内置的权限。这个后端无法防范暴力破解攻击，因为没有提供频率限制机制。你可以在自定义的权限核准后端中实现这一机制，或者使用多数 Web 服务器提供的机制。AUTHENTICATION_BACKENDS 罗列元素的顺序是重要的，如果用户名和密码能通过多个后端的验证，首次发现匹配后就会停止。如果后端抛出 PermissionDenied 异常，身份验证立即失败，Django 不再使用后面的后端检查。

通过身份验证后，Django 在用户的会话中存储通过验证的是哪个后端。在会话持续的时间内，只要需要访问通过验证的用户，就使用会话中的后端。这其实意味着，每个会话会缓存身份验证源，所以如果修改了 AUTHENTICATION_BACKENDS，若想使用其他方法重新验证用户的身份，要清空会话数据。为此，一个简单的方法是调用 Session.objects.all().delete()。

11.10.3 编写一个身份验证后端

身份验证后端通过类定义，必须实现两个方法，即 get_user(user_id) 和 authenticate(**credentials)，此外还有一些可选的权限核准方法。get_user 的参数 user_id 可以是用户名和数据库中的 ID 等，不过必须是 User 对象的主键。get_user 的返回值是一个 User 对象。authenticate 的参数是通过关键字参数指定的凭据。多数时候，authenticate 方法是下面这样：

```
class MyBackend(object):
```

```
    def authenticate(self, username=None, password=None):
        # 检查用户名和密码，返回一个 User 对象
        ...
```

不过，也可以验证令牌，如下所示：

```
class MyBackend(object):
    def authenticate(self, token=None):
        # 检查令牌，返回一个 User 对象
        ...
```

不管怎样，authenticate 都应该检查传入的凭据，通过检查时返回对应的 User 对象。如果检查失败，应该返回 None。Django 管理后台与本章开头描述的 User 对象紧密耦合。

目前，最佳处理方式是为后端（如 LDAP 目录、外部 SQL 数据库，等等）里的各个用户创建一个 Django User 对象。你可以编写一个脚本事先创建好，也可以在用户首次登录时让 authenticate 方法创建。

下面的示例后端验证 settings.py 文件中定义的用户名和密码，在首次验证用户的身份时创建 Django User 对象：

```
from django.conf import settings
from django.contrib.auth.models import User, check_password

class SettingsBackend(object):
    """
    Authenticate against the settings ADMIN_LOGIN and ADMIN_PASSWORD.

    Use the login name, and a hash of the password. For example:

    ADMIN_LOGIN = 'admin'
    ADMIN_PASSWORD = 'sha1$4e987$afbcf42e21bd417fb71db8c66b321e9fc33051de'
    """

    def authenticate(self, username=None, password=None):
        login_valid = (settings.ADMIN_LOGIN == username)
        pwd_valid = check_password(password, settings.ADMIN_PASSWORD)
        if login_valid and pwd_valid:
            try:
                user = User.objects.get(username=username)
            except User.DoesNotExist:
                # 新建用户
                # 注意，可以把密码设为任何值，因为不会检查它
                # 检查的是 settings.py 文件中定义的密码
                user = User(username=username, password='password')
                user.is_staff = True
                user.is_superuser = True
                user.save()
            return user
        return None

    def get_user(self, user_id):
        try:
            return User.objects.get(pk=user_id)
```

```python
        except User.DoesNotExist:
            return None
```

11.10.4 在自定义后端中核准权限

自定义的权限核准后端可以提供自己的权限。用户模型会把权限查找函数（get_group_permissions()、get_all_permissions()、has_perm() 和 has_module_perms()）委托给实现了这些函数的身份验证后端。赋予用户的权限将是所有后端返回的全部权限的超集。即，Django 为用户赋予每个后端定义的权限。

如果后端在 has_perm() 或 has_module_perms() 函数中抛出 PermissionDenied 异常，权限核准立即失败，Django 不会检查后面的后端。前面那个示例后端可以轻易实现管理员权限：

```python
class SettingsBackend(object):
    ...
    def has_perm(self, user_obj, perm, obj=None):
        if user_obj.username == settings.ADMIN_LOGIN:
            return True
        else:
            return False
```

这为上述示例中获准访问的用户赋予完整的权限。注意，除了 User 模型相关函数的那些参数之外，后端的权限核准函数还都接受用户对象（可以是匿名用户）为参数。

django/contrib/auth/backends.py 文件中的 ModelBackend 类完整实现了权限核准后端。这是默认的后端，多数时候查询的是 auth_permission 表。如果只想自定义后端 API 的部分行为，可以利用 Python 继承，定义 ModelBackend 的子类，而不是在自定义的后端中实现全部 API。

11.10.5 核准匿名用户的权限

匿名用户是未验证身份的用户，即没有提供有效的身份验证信息。然而，这并不意味着匿名用户没权限做任何事。基本上，多数网站允许匿名用户浏览大部分页面，而且还有一些网站允许匿名用户发表评论等。

Django 的权限框架无法存储匿名用户的权限。然而，传给身份验证后端的用户对象可能是 django.contrib.auth.models.AnonymousUser 的实例，因此可以为匿名用户指定自定义的权限核准行为。

这对可复用应用的开发者来说尤其有用，他们可以把所有权限核准交给身份验证后端，而不用控制匿名访问。

11.10.6 核准未激活用户的权限

未激活的用户是已经验证了身份、但是 is_active 属性被设为 False 的用户。然而，这并不意味着未激活的用户没权限做任何事。例如，允许他们激活自己的账户。

权限系统对匿名用户的支持允许匿名用户有的权限而未激活的用户没有。别忘了在后端的权限核准方法中测试用户的 is_active 属性。

11.10.7 处理对象的权限

Django 的权限框架为对象的权限建立了基础，但是没有在核心中实现。这意味着，对对象权限的检查始终返回 False 或空列表（取决于检查的对象）。核准对象权限的每个方法都接受关键字参数 obj 和 user_obj，可以返回相应的对象级权限。

11.11 自定义权限

若想为指定的模型对象自定义权限，使用模型中 Meta 类的 permissions 属性。下述 Task 模型自定义了三个权限，分别是用户可在应用程序内对 Task 实例能做或不能做的操作：

```
class Task(models.Model):
    ...
    class Meta:
        permissions = (
            ("view_task", "Can see available tasks"),
            ("change_task_status", "Can change the status of tasks"),
            ("close_task", "Can remove a task by setting its status as
              closed"),
        )
```

这么做的唯一作用是在执行 manage.py migrate 命令时创建这些额外的权限。用户尝试访问应用程序提供的功能时（查看任务、修改任务的状态或关闭任务），要在代码中检查这些权限的值。接着上例，下述代码检查用户可不可以查看任务：

```
user.has_perm('app.view_task')
```

11.12 扩展现有的 User 模型

除了把默认的 User 模型替换掉，还有两种扩展它的方式。如果你想做的修改只是行为上的，不需要修改数据库中存储的数据，可以基于 User 创建一个代理模型。这样能得到代理模型提供的各个功能，如默认的排序、自定义管理器或自定义模型方法。

如果想存储关于用户的额外信息，可以与另一个模型建立一对一关系，把信息存储在那个模型的字段中。这种通过一对一关系连接的模型通常称为个人资料模型（profile model），因为它可能存储着与身份验证无关的信息。比如说，可以创建下述 Employee 模型：

```
from django.contrib.auth.models import User

class Employee(models.Model):
    user = models.OneToOneField(User)
    department = models.CharField(max_length=100)
```

假如员工 Fred Smith 既属于 User 模型，也属于 Employee 模型，可以使用 Django 标准的相关模型约定访问额外的信息：

```
>>> u = User.objects.get(username='fsmith')
>>> freds_department = u.employee.department
```

若想把个人资料模型中的字段添加到管理后台中的用户页面中，在应用的 admin.py 文件中定义一个 InlineModelAdmin 类（这里使用的是 StackedInline），然后把它添加到 UserAdmin 类中，再注册到 User 类上：

```
from django.contrib import admin
from django.contrib.auth.admin import UserAdmin
from django.contrib.auth.models import User

from my_user_profile_app.models import Employee

# 为 Employee 模型定义一个内联管理后台描述符
```

```python
# 它的行为有点像单例
class EmployeeInline(admin.StackedInline):
    model = Employee
    can_delete = False
    verbose_name_plural = 'employee'

# 定义一个 UserAdmin 的子类
class UserAdmin(UserAdmin):
    inlines = (EmployeeInline, )

# 重新注册 UserAdmin
admin.site.unregister(User)
admin.site.register(User, UserAdmin)
```

个人资料模型没什么特殊的，就是普通的 Django 模型，只是碰巧与 `User` 模型有一对一关系。因此，创建用户时不会自动创建对应的个人资料，不过可以通过 `django.db.models.signals.post_save` 信号创建或更新相关的模型。

注意，通过相关的模型检索数据时有额外的查询或联结，某些情况下替换 `User` 模型或添加额外的字段可能更好。然而，项目中的应用对默认 `User` 模型的现有链接可能会调整额外的数据库负载。

11.13 替换成自定义的 User 模型

对有些项目来说，Django 内置的 `User` 模型可能无法满足身份验证的需求。例如，较之用户名，有些网站更适合使用电子邮件地址作为用户的唯一标识符。Django 允许覆盖默认的 `User` 模型，方法是把 `AUTH_USER_MODEL` 设置的值设为自定义的模型：

```python
AUTH_USER_MODEL = 'books.MyUser'
```

点号前面是 Django 应用的名称（必须在 `INSTALLED_APPS` 设置中列出），后面是想用作 `User` 模型的 Django 模型名。

> **提醒**
>
> 修改 `AUTH_USER_MODEL` 设置对 Django 项目有重大的影响，尤其是数据库结构。如果在运行迁移之后修改 `AUTH_USER_MODEL`，必须自己动手更新数据库，因为很多数据库表的关系受到了影响。除非有特别好的理由，否则不要修改 `AUTH_USER_MODEL`。

尽管我在上面做出了提醒，但是要知道 Django 是完全支持自定义用户模型的。具体做法超出了本书范畴。兼容管理后台的完整示例，以及对如何自定义用户模型的全面说明，参见 Django Project 网站。

11.14 接下来

本章学习了 Django 的用户身份验证系统、内置的身份验证工具，以及大量可定制的功能。下一章探讨算得上是创建和维护强健的应用最为重要的一个工具——自动化测试。

第 12 章 测试 Django 应用程序

12.1 测试简介

与所有成熟的框架一样，Django 也内置了单元测试功能。单元测试是一种软件测试过程，测试的是软件应用程序的独立单元，确保能做预期中的事情。

单元测试分为不同的层级，可以测试单个方法，看它能不能返回正确的值以及能否处理正确的数据，也可以测试整个方法组件，确保一系列用户输入能得到所需的结果。

在单元测试背后，有四个基本的概念：

1. *测试固件*（test fixture），执行测试所需的设置。包含数据库、示例数据集合服务器搭建。测试固件可能还包括测试完毕后执行的清理操作。
2. *测试用例*（test case），测试的基本单元。测试用例检查指定的输入是否能得到预期的结果。
3. *测试组件*（test suite），一系列测试用例或其他测试组件，作为一个整体执行。
4. *测试运行程序*（test runner），负责执行测试并把结果反馈给用户的软件程序。

软件测试是一门很深的学问，涉及众多知识，本章只是对单元测试做个简略介绍。网上有大量关于软件测试理论和方法的资源，我建议你研读一下。若想深入了解 Django 采用的单元测试方法，请访问 Django Project 网站。

12.2 自动化测试简介

12.2.1 自动化测试是什么

你可能没注意到，本书前面已经编写了测试代码。在 Django shell 中确认函数是否可用，或者看给定的输入能得到什么输出，这就是在测试你的代码。例如，在第 2 章，我们把一个字符串传给期待整数的视图，那个视图抛出 `TypeError` 异常。

测试是应用程序开发的常规组成部分，不过自动化测试的不同之处是，测试工作由系统代劳。编写一系列测试之后，如果修改了应用，可以检查代码是否还能像之前那样工作，而不用每次都花时间自己动手测试。

12.2.2 为什么测试

如果你所做的 Django 编程工作只是创建一个像本书中那样的简单应用程序，确实，你无需知道如何编写自动化测试。但是，如果你想成为专业的程序员，想开发更复杂的项目，必须要知道如何编写自动化测试。

自动化测试有以下优点：

- *节省时间*。手动测试大型应用程序各组件之间庞杂的交互浪费时间，而且易于出错。自动化测试能节省时间，让你专注编程。
- *避免出问题*。测试能强调代码的内部运作，让你发现什么地方存在错误。
- *看起来专业*。专业人士都编写测试。Django 最初的开发者之一 Jacob Kaplan-Moss 说："没有测试的代

码先天不足。”

- 增进团队协作。测试能保证同事不会意外破坏你的代码（你自己也不会不小心破坏别人的代码）。

12.3 基本的测试策略

编写测试的方式很多。有些程序员遵守一个称为*测试驱动开发*（Test-Driven Development，TDD）的准则，先写测试后写代码。看上去这可能有点违背直觉，但是其实多数人通常都是这么做的：先描述问题，再编写解决问题的代码。

测试驱动开发为 Python 测试用例的编写定下了基调。通常，测试新手会先编写代码，然后再判断是否应该编写测试。其实，早点编写测试或许更好，但是何时着手都不晚。

12.4 编写一个测试

在开始编写首个测试之前，我们先在 Book 模型中引入一个缺陷。

比如说我们决定在 Book 模型中自定义一个方法，指明一本书是不是最近出版的。Book 模型可以像这样定义：

```
import datetime
from django.utils import timezone

from django.db import models

# ... #

class Book(models.Model):
    title = models.CharField(max_length=100)
    authors = models.ManyToManyField(Author)
    publisher = models.ForeignKey(Publisher)
    publication_date = models.DateField()

    def recent_publication(self):
        return self.publication_date >= timezone.now().date() - datetime.timedelta(weeks=8)

    # ... #
```

首先，导入两个模块：Python 标准库中的 `datetime` 和 `django.utils` 中的 `timezone`。这是计算日期所需的。然后，在 Book 模型中定义一个名为 `recent_publication` 的方法，计算 8 周前的日期，当图书的出版日期晚于那一天时返回 `True`。

下面我们打开交互式 shell，测试一下这个新方法：

```
python manage.py shell
```

```
>>> from books.models import Book
>>> import datetime
>>> from django.utils import timezone
>>> book = Book.objects.get(id=1)
>>> book.title
'Mastering Django: Core'
>>> book.publication_date
```

```
datetime.date(2016, 5, 1)
>>>book.publication_date >= timezone.now().date() - datetime.timedelta(weeks=8)
True
```

目前来看没什么问题，我们先导入 Book 模型，然后检索一本书。今天是 2016 年 6 月 11 日，我在数据库中填写的出版日期是 5 月 1 日，在 8 周的范围内，因此这个方法正确地返回了 True。

显然，你自己测试时要修改出版日期，这样测试才能得到上述结果。

现在，我们把出版日期设为未来的某一天，比如 9 月 1 日，看看会发生什么：

```
>>> book.publication_date
datetime.date(2016, 9, 1)
>>>book.publication_date >= timezone.now().date() - datetime.timedelta(weeks=8)
True
```

不妙，这里显然有问题。你应该很快就能在逻辑中发现错误——任何晚于 8 周前那一天的日期都返回 True，包括未来的日期。

当然这是编造的示例，不过撇开这一点不管，下面来为这个有问题的逻辑编写一个测试。

12.4.1 编写测试

使用 Django 的 startapp 命令创建 books 应用时，它在应用的目录中创建了一个名为 tests.py 的文件。这个文件用于保存 books 应用的测试。打开那个文件，编写一个测试：

```
import datetime
from django.utils import timezone
from django.test import TestCase
from .models import Book

class BookMethodTests(TestCase):

    def test_recent_pub(self):
        """
        recent_publication() should return False for future publication
        dates.
        """

        futuredate = timezone.now().date() + datetime.timedelta(days=5)
        future_pub = Book(publication_date=futuredate)
        self.assertEqual(future_pub.recent_publication(), False)
```

这段代码十分简单，基本上与前面在 Django shell 中所做的一样。唯一的不同是，现在我们把测试代码封装在一个类中，而且创建了一个断言（assertion），使用未来的日期测试 recent_publication() 方法。

本章后文会详细介绍测试类和 assertEqual 方法，现在我们的目的只是了解测试的基本结构。

12.4.2 运行测试

写好测试之后要运行。幸好，这并不难。打开终端，输入下述命令：

```
python manage.py test books
```

稍等片刻，Django 应该输出类似下面的内容：

```
Creating test database for alias 'default'...
F
======================================================================
FAIL: test_recent_pub (books.tests.BookMethodTests)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\Users\Nigel\ ... mysite\books\tests.py", line 25, in test_recent_pub
    self.assertEqual(future_pub.recent_publication(), False)
AssertionError: True != False

----------------------------------------------------------------------
Ran 1 test in 0.000s

FAILED (failures=1)
Destroying test database for alias 'default'...
```

上述过程详述如下：

- `python manage.py test books` 命令在 books 应用中查找测试。
- 找到 `django.test.TestCase` 类的一个子类。
- 为测试创建一个特殊的数据库。
- 查找名称以“test”开头的方法。
- 在 `test_recent_pub` 中，创建一个 `Book` 实例，把 `publication_date` 属性设为 5 天后的日期。
- 调用 `assertEqual()` 方法，发现那个实例的 `recent_publication()` 方法返回 `True`，而它本该返回 `False`。
- 报告失败的测试，以及是哪一行导致失败的。注意，如果使用 *nix 系统或 Mac，文件路径有所不同。

以上就是对 Django 测试的简单介绍。本章开头说过，测试是一门很深的学问，涉及众多知识，而且对程序员的职业发展十分重要。在简短的一章内容里无法面面俱到，因此我建议你阅读本章提到的资料和 Django 文档。

本章余下的内容介绍 Django 提供的各种测试工具。

12.5 测试工具

Django 为编写测试提供了一些便利的工具。

12.5.1 测试客户端

测试客户端是一个 Python 类，伪装成一个 Web 浏览器，以便通过编程的方式测试 Django 应用程序的视图和交互。测试客户端能做的事情有：

- 模拟对 URL 的 `GET` 和 `POST` 请求，并监视响应，这包括低层的 HTTP 信息（首部和状态码）和页面内容。
- 跟踪重定向过程（如果有的话），检查这个过程中每一步的 URL 和状态码。
- 测试指定的请求由指定的 Django 模板渲染，而且模板上下文中包含特定的值。

注意，这个测试客户端的目的不是取代 Selenium 或其他操作浏览器的框架。Django 的测试客户端关注的点不同。

简单来说：

- Django 的测试客户端用于确认是否渲染了正确的模板，而且为模板传入了正确的上下文数据。
- 操作浏览器的框架（如 Selenium）用于测试渲染得到的网页中的内容和行为，尤其是使用 JavaScript 实现的功能。

Django 也为那些框架提供了特别支持，详情参见 `LiveServerTestCase` 一节。全面的测试组件应该二者结合。若想查看详细的测试客户端示例，请访问 Django Project 网站。

12.5.2 内置的测试用例类

Python 常规的单元测试类扩展自 `unittest.TestCase` 类。Django 为这个基类提供了一些扩展。

`SimpleTestCase`

扩展 `unittest.TestCase`，提供下述基本功能：

- 保存和恢复 Python 提醒状态。
- 添加一些有用的断言，如：
  - 检查可调用的对象抛出特定的异常。
  - 测试表单字段渲染和错误处理。
  - 测试 HTML 响应，检查存在或没有指定的片段。
  - 确认模板已经或尚未用于生成指定的响应内容。
  - 确认应用做了 HTTP 重定向。
  - 不严格测试两个 HTML 片段是否相同、不同或有包含关系。
  - 不严格测试两个 XML 片段是否相同或不同。
  - 不严格测试两个 JSON 片段是否相同。
- 使用自定义的设置运行测试。
- 使用测试客户端。
- 自定义测试-时间 URL 映射。

`TransactionTestCase`

Django 的 `TestCase` 类（下文说明）利用数据库事务提速每个测试开头把数据库还原为已知状态的过程。然而，这么做的后果是，有些数据库行为在 `TestCase` 类中无法测试。

此时，应该使用 `TransactionTestCase`。它的行为与 `TestCase` 一样，不过把数据库还原成已知状态的方式不同，而且测试代码能测试提交和回滚的效果：

- `TransactionTestCase` 在测试运行完毕后还原数据库，把所有表删除。`TransactionTestCase` 可能会调用提交或回滚操作，然后在数据库中监视效果。
- 而 `TestCase` 在测试完毕后不删除表。它把测试代码放在一个数据库事务中，在测试的末尾回滚。这么

做的目的是确保把数据库恢复到最初状态。

`TransactionTestCase` 继承自 `SimpleTestCase`。

`TestCase`

这个类提供了一些额外的功能，在测试网站时用得到。把常规的 `unittest.TestCase` 转换成 Django 的 `TestCase` 很容易：把测试的基类由 `unittest.TestCase` 改为 `django.test.TestCase` 即可。标准的 Python 单元测试所具有的功能仍然可用，此外还增加了一些功能：

- 自动加载固件。
- 把测试包装到两个嵌套的 `atomic` 块中：一个针对整个类，一个针对各个测试。
- 创建 `TestClient` 实例。
- Django 专有的测试断言，如用于测试重定向和表单错误的断言。

`TestCase` 继承自 `TransactionTestCase`。

`LiveServerTestCase`

`LiveServerTestCase` 的作用基本与 `TransactionTestCase` 相同，不过有个额外功能：测试前在后台启动线上 Django 服务器，测试后再关闭。此时使用的是自动化测试客户端（Selenium）而不是 Django 的模拟客户端，因此可以在浏览器中执行功能测试，模拟真实的用户操作。

12.5.3 测试用例的功能

默认的测试客户端

*`TestCase` 类中的每个测试都能访问 Django 测试客户端：`self.client`。这个客户端在每个测试中重新创建，因此不用担心会带有其他测试的状态（如 cookie）。这意味着，不用在每个测试中实例化 `Client`：

```
import unittest
from django.test import Client

class SimpleTest(unittest.TestCase):
    def test_details(self):
        client = Client()
        response = client.get('/customer/details/')
        self.assertEqual(response.status_code, 200)

    def test_index(self):
        client = Client()
        response = client.get('/customer/index/')
        self.assertEqual(response.status_code, 200)
```

而是通过 `self.client` 引用：

```
from django.test import TestCase

class SimpleTest(TestCase):
    def test_details(self):
        response = self.client.get('/customer/details/')
```

```
        self.assertEqual(response.status_code, 200)

    def test_index(self):
        response = self.client.get('/customer/index/')
        self.assertEqual(response.status_code, 200)
```

加载固件

如果数据库中没有数据，对以数据库为后台的网站来说，测试用例没有多大用。Django 自定义的 TransactionTestCase 类提供了一种方式，能简化把测试数据存入数据库的过程：加载固件。固件是一系列数据，Django 知道如何将其导入数据库。比如说，网站有一些账户，你可以创建包含虚拟账户的固件，在测试中填充数据库。

创建固件最简单的方式是使用 manage.py dumpdata 命令。不过前提是数据库中已经有一些数据。（详情参见 dumpdata 命令的文档。）

创建好固件，并将其放到 INSTALLED_APPS 中某个应用的 fixtures 目录里之后，在 django.test.TestCase 的子类中设定类属性 fixtures：

```
from django.test import TestCase
from myapp.models import Animal

class AnimalTestCase(TestCase):
    fixtures = ['mammals.json', 'birds']

    def setUp(self):
        # 像之前那样定义测试
        call_setup_methods()

    def testFluffyAnimals(self):
        # 一个使用固件的测试
        call_some_test_code()
```

加载固件的过程如下：

- 在各个测试用例之前，也在 setUp() 运行之前，Django 清理数据库，把数据库还原成执行 migrate 命令之后的状态。
- 然后，加载所有具名固件。这里，Django 会先加载名为 mammals 的 JSON 固件，再加载名为 birds 的固件。

关于定义和加载固件的详情，参阅 loaddata 命令的文档。测试用例中的每个测试都会重复上述清理和加载过程，因此可以保证的是，一个测试的输出对另一个测试没有影响，而且执行测试的顺序也无关。默认情况下，固件只加载到 default 数据库中。如果使用多个数据库，而且设定了 multi_db=True，固件会加载到所有数据库中。

覆盖设置

> **提醒**
>
> 下述函数仅用于在测试中临时修改设置。别直接处理 django.conf.settings，因为修改之后无法还原成原来的值。

settings()

测试中经常需要临时修改设置，然后在测试完毕后恢复原值。为此，Django 提供了一个标准的 Python 上下文管理器（参见 PEP 343），名为 settings()。用法如下：

```
from django.test import TestCase

class LoginTestCase(TestCase):

    def test_login(self):

        # 首先检查默认的行为
        response = self.client.get('/sekrit/')
        self.assertRedirects(response, '/accounts/login/?next=/sekrit/')

        # 然后覆盖 LOGIN_URL 设置
        with self.settings(LOGIN_URL='/other/login/'):
            response = self.client.get('/sekrit/')
            self.assertRedirects(response, '/other/login/?next=/sekrit/')
```

这个示例在 with 块中覆盖 LOGIN_URL 设置，然后恢复原值。

modify_settings()

重新定义值为列表的设置很不方便。实际使用中，添加或删除元素往往就行了。modify_settings() 上下文管理器能简化这个过程：

```
from django.test import TestCase

class MiddlewareTestCase(TestCase):

    def test_cache_middleware(self):
        with self.modify_settings(MIDDLEWARE_CLASSES={
            'append': 'django.middleware.cache.FetchFromCacheMiddleware',
            'prepend': 'django.middleware.cache.UpdateCacheMiddleware',
            'remove': [
               'django.contrib.sessions.middleware.SessionMiddleware',
               'django.contrib.auth.middleware.AuthenticationMiddleware',
               'django.contrib.messages.middleware.MessageMiddleware',
            ],
        }):
            response = self.client.get('/')
            # ...
```

每次操作可以提供一组值或一个字符串。如果列表中已有提供的值，append 和 prepend 无效果；如果列表中没有提供的值，remove 也没效果。

override_settings()

如果想在测试方法中覆盖设置，使用 Django 提供的 override_settings() 装饰器（参见 PEP 318）。用法如下：

```
from django.test import TestCase, override_settings
```

```
class LoginTestCase(TestCase):

    @override_settings(LOGIN_URL='/other/login/')
    def test_login(self):
        response = self.client.get('/sekrit/')
        self.assertRedirects(response, '/other/login/?next=/sekrit/')
```

这个装饰器还可运用到测试用例类上：

```
from django.test import TestCase, override_settings

@override_settings(LOGIN_URL='/other/login/')
class LoginTestCase(TestCase):

    def test_login(self):
        response = self.client.get('/sekrit/')
        self.assertRedirects(response, '/other/login/?next=/sekrit/')
```

`modify_settings()`

同样，Django 还提供了 `modify_settings()` 装饰器：

```
from django.test import TestCase, modify_settings

class MiddlewareTestCase(TestCase):

    @modify_settings(MIDDLEWARE_CLASSES={
        'append': 'django.middleware.cache.FetchFromCacheMiddleware',
        'prepend': 'django.middleware.cache.UpdateCacheMiddleware',
    })
    def test_cache_middleware(self):
        response = self.client.get('/')
        # ...
```

这个装饰器也可以运用到测试用例类上：

```
from django.test import TestCase, modify_settings

@modify_settings(MIDDLEWARE_CLASSES={
    'append': 'django.middleware.cache.FetchFromCacheMiddleware',
    'prepend': 'django.middleware.cache.UpdateCacheMiddleware',
})
class MiddlewareTestCase(TestCase):

    def test_cache_middleware(self):
        response = self.client.get('/')
        # ...
```

覆盖设置时要记得处理使用缓存或类似功能的情况，因为此时即使修改设置，状态依然不变。Django 提供了 `django.test.signals.setting_changed` 信号，你可以注册回调，在状态改变时清理或还原状态。

断言

Python 中常规的 unittest.TestCase 类实现了 assertTrue() 和 assertEqual() 等断言方法，Django 自定义的 TestCase 类提供了一些对测试 Web 应用有用的断言方法：

- assertRaisesMessage：断言执行的可调用对象抛出了异常，而且消息为 expected_message。
- assertFieldOutput：断言表单字段的行为在多种输入下是正确的。
- assertFormError：断言渲染表单时某个字段抛出指定列表中的错误。
- assertFormsetError：断言渲染 formset 时抛出指定列表中的错误。
- assertContains：断言 Response 实例的状态码为 status_code，而且 text 出现在响应内容中。
- assertNotContains：断言 Response 实例的状态码为 status_code，但是 text 未出现在响应内容中。
- assertTemplateUsed：断言使用指定名称的模板渲染响应。模板的名称是一个字符串，例如 'admin/index.html'。
- assertTemplateNotUsed：断言未使用指定名称的模板渲染响应。
- assertRedirects：断言响应是状态码为 status_code 的重定向，重定向到 expected_url（包含 GET 数据），而且收到的最终页面的状态码是 target_status_code。
- assertHTMLEqual：断言字符串 html1 和 html2 相同。基于 HTML 语义比较。比较时会考虑下述情况：
  - 忽略 HTML 标签前后的空白。
  - 不管是何种空白，都认为相同。
  - 起始标签隐式关闭，例如外层标签关闭，或者 HTML 文档结束。
  - 空标签与自关闭版本相同。
  - HTML 元素属性的顺序无关紧要。
  - 没有值的属性与名称和值一样的属性相同（详见示例）。
- assertHTMLNotEqual：断言字符串 html1 和 html2 不同。基于 HTML 语义比较。详情参见 assertHTMLEqual()。
- assertXMLEqual：断言字符串 xml1 和 xml2 相同。基于 XML 语义比较。与 assertHTMLEqual() 类似，比较的是解析后的内容，因此只考虑语义区别，而不是句法区别。
- assertXMLNotEqual：断言字符串 xml1 和 xml2 不同。基于 XML 语义比较。详情参见 assertXMLEqual()。
- assertInHTML：断言 HTML 片段 needle 包含在 haystack 中。
- assertJSONEqual：断言 JSON 片段 raw 和 expected_data 相同。
- assertJSONNotEqual：断言 JSON 片段 raw 和 expected_data 不同。
- assertQuerysetEqual：断言查询集合 qs 返回 values 中指定的一组值。使用 transform 函数比较 qs 和 values 的内容。默认情况下，这意味着比较的是每个值的字符串表示形式（repr()）。
- assertNumQueries：断言调用 func 时传入 *args 和 **kwargs，执行的数据库查询次数为 num。

12.5.4 电子邮件服务

如果有 Django 视图使用 Django 的电子邮件功能发送电子邮件，你可能不想每次运行那个视图的测试时都发送电子邮件。鉴于此，Django 的测试运行程序自动把 Django 发出的邮件重定向到一个模拟的发件箱中。这样能全方位测试邮件发送，从发送的数量到每封邮件的内容，而不用真的发送邮件。测试运行程序在背后所

做的是，把常规的电子邮件后端替换成测试后端。（别担心，这对 Django 外部的邮件发送程序没有任何影响，例如设备中的邮件服务器。）

运行测试时，发出的电子邮件保存在 django.core.mail.outbox 中。这是一个由 EmailMessage 实例构成的列表。outbox 是个特殊的属性，仅当使用 locmem 电子邮件后端时才创建。它不是 django.core.mail 模块的常规组成部分，无法直接导入。下述代码展示如何正确地访问这个属性。下述示例测试检查 django.core.mail.outbox 的长度和内容：

```
from django.core import mail
from django.test import TestCase

class EmailTest(TestCase):
    def test_send_email(self):
        # 发送邮件
        mail.send_mail('Subject here', 'Here is the message.',
            'from@example.com', ['to@example.com'],
            fail_silently=False)

        # 断言发出了一封邮件
        self.assertEqual(len(mail.outbox), 1)

        # 验证第一封邮件的主题是正确的
        self.assertEqual(mail.outbox[0].subject, 'Subject here')
```

如前所述，*TestCase 中的每个测试运行前都会清空测试发件箱。如果想手动清空，把 mail.outbox 设为一个空列表：

```
from django.core import mail

# 清空测试发件箱
mail.outbox = []
```

12.5.5 管理命令

管理命令可以使用 call_command() 函数测试。输出可以重定向到一个 StringIO 实例中：

```
from django.core.management import call_command
from django.test import TestCase
from django.utils.six import StringIO

class ClosepollTest(TestCase):
    def test_command_output(self):
        out = StringIO()
        call_command('closepoll', stdout=out)
        self.assertIn('Expected output', out.getvalue())
```

12.5.6 跳过测试

如果事先知道特定的情况下测试将失败，可以使用 unittest 库提供的 @skipIf 和 @skipUnless 装饰器跳过测试。假如测试需要特定的可选库才能通过，可以使用 @skipIf 装饰测试用例。然后，测试运行程序会报告测试未执行，并指明原因，而不是让测试失败，或者将其忽略。

12.6 测试数据库

需要数据库的测试（即模型测试）不会使用生产数据库，Django 会为测试单独创建空数据库。不管测试通过还是失败，测试数据库都在全部测试执行完毕后销毁。如果不想销毁测试数据库，运行 `test` 命令时指定 `--keepdb` 旗标。这样，在两次运行测试之间测试数据库会留存下来。

如果测试数据库不存在，首先新建。为了保证模式最新，还会运行迁移。默认情况下，测试数据库的名称是在 `DATABASES` 设置中的 `NAME` 选项前加 `test_`。如果使用 SQLite 数据库引擎，测试默认使用内存中的数据库（即，数据库在内存中创建，完全不用文件系统）。

如果想使用其他数据库名称，在 `TEST` 设置中为 `DATABASES` 设置列出的各个数据库指定 `NAME` 选项。使用 PostgreSQL 时，`USER` 还要有内置 `postgres` 数据库的读权限。对同一个数据库来说，测试运行程序始终使用设置文件中的数据库设置：`ENGINE`、`USER`、`HOST`，等等。测试数据库由 `USER` 指定的用户创建，因此要确保指定的用户有足够的权限在系统中新建数据库。

12.7 使用其他测试框架

显然，`unittest` 不是唯一的 Python 测试框架。虽然 Django 不直接支持其他测试框架，但是却为调用其他框架编写的测试提供了方式，让 Django 认为那就是普通的 Django 测试。

执行 `./manage.py test` 命令时，Django 根据 `TEST_RUNNER` 设置决定接下来做什么。`TEST_RUNNER` 默认指向 `django.test.runner.DiscoverRunner`。这个类定义 Django 默认的测试行为，包括：

1. 执行全局的测试前准备工作。
2. 在当前目录中查找名称匹配 `test*.py` 模式的测试文件。
3. 创建测试数据库。
4. 运行迁移，把模型和初始数据填充到测试数据库中。
5. 运行找到的测试。
6. 销毁测试数据库。
7. 执行全局的测试后清理工作。

如果你自己定义了测试运行程序类，而且把 `TEST_RUNNER` 指向它，每次执行 `./manage.py test` 命令时，Django 都会使用那个测试运行程序。

如此以来便可以使用任何 Python 测试框架，或者调整 Django 执行测试的过程，满足特定的需求。

关于这个话题的详细说明，参见 Django Project 网站。

12.8 接下来

现在你知道如何为自己的 Django 项目编写测试了，下面我们换个重要话题，讨论如何把项目变成真正的线上网站，即把 Django 项目部署到 Web 服务器上。

第 13 章 部署 Django 应用程序

本章介绍构建 Django 应用程序的最后一个重要步骤：把应用程序部署到生产服务器。

如果你一直跟着书中的示例做，那就用过 `runserver` 命令了。这个命令简化了相关操作，不用你费心搭建 Web 服务器。但是，`runserver` 只适合在本地设备中做开发，不能在公开 Web 中使用。

若想部署 Django 应用程序，要使用工业级 Web 服务器，例如 Apache。本章将说明具体怎么做。不过，首先要告诉你上线之前需要在代码基中做的事情。

13.1 为上线做好准备

13.1.1 部署点检表

互联网危机四伏。部署 Django 项目之前，应该花点时间审查设置，考虑安全、性能和运维。

Django 包含很多安全措施，有些是内置的，而且始终启用，有些则是可选的，因为并非始终适用，或者对开发不便。例如，强制使用 HTTPS 可能不适合所有网站，在本地开发中也没有意义。

性能优化也是为了便利而有所牺牲的一方面。例如，缓存对生产环境很有用，但是在本地开发中就没多大用处。错误报告需求也有诸多不同情况。接下来介绍的点检表涉及以下几方面的设置：

- 必须正确设置，以便提供预期的安全级别
- 在各个环境中本该不同
- 启用可选的安全措施
- 启用性能优化
- 提供错误报告

这些设置中很多是敏感的，应该视作机密。如果想发布项目的源码，常见的做法是公开适合开发使用的设置，而在生产环境中使用隐私的设置模块。下面所述的检查项目中，有些可以在 `check` 命令中指定 `--deploy` 选项自动检查。一定要像 `--deploy` 选项的文档中所说的那样使用这个命令检查生产环境的设置文件。

13.2 关键设置

13.2.1 SECRET_KEY

密钥必须是一长串随机值，而且必须保密。

千万不能在别处使用生产环境的密钥，而且不能提交到源码控制系统中。这么做是为了降低攻击者盗取密钥的机会。别在设置模块中硬编码密钥，应该考虑从环境变量中加载：

```
import os
SECRET_KEY = os.environ['SECRET_KEY']
```

或者从文件中加载：

```python
with open('/etc/secret_key.txt') as f:
    SECRET_KEY = f.read().strip()
```

13.2.2 DEBUG

一定不能在生产环境启用调试模式。

我们在第 1 章创建项目时，`django-admin startproject` 命令创建的 `settings.py` 文件把 `DEBUG` 设为 `True`。Django 的很多内部组件检查这个设置，如果启用了调试模式，行为有所不同。

假如把 `DEBUG` 设为 `True` 了，那么：

- 所有数据库查询以 `django.db.connection.queries` 对象的形式存储在内存中。可以想象，这样很耗内存！
- 404 错误使用 Django 特殊的 404 页面渲染（参见第 3 章），而不是返回 404 响应。那个页面可能包含敏感信息，因此不应该公开在互联网中显示。
- Django 应用程序中任何未捕获的异常（从基本的 Python 句法错误、数据库错误，到模板句法错误）都使用 Django 精美的错误页面渲染。你可能见过这个页面，而且很喜欢。这个页面包含的敏感信息比 404 页面还多，决不能公开显示。

简单来说，把 `DEBUG` 设为 `True` 是告诉 Django，只有信任的开发者在使用网站。互联网鱼龙混杂，因此准备部署应用程序时，首先应该把 `DEBUG` 设为 `False`。

13.3 各环境专用的设置

13.3.1 ALLOWED_HOSTS

设定 `DEBUG = False` 之后，如果不为 `ALLOWED_HOSTS` 设定合适的值，Django 根本无法运转。这个设置的作用是防范某些 CSRF 攻击，因此必须设定。如果使用泛域名，必须自行验证 HTTP `Host` 首部，或者确保没有这方面的漏洞。

13.3.2 缓存

如果使用缓存，开发环境和生产环境使用的连接参数可能不同。缓存服务器往往不严格验证身份，因此要确保只接受来自应用服务器的连接。如果使用 Memcached，请考虑缓存会话，以提升性能。

13.3.3 数据库

数据库连接参数在开发环境和生产环境中可能不同。数据库密码是敏感信息，应该像 `SECRET_KEY` 那样保护起来。为了最大限度的增强安全，确保数据库服务器只接受来自应用服务器的连接。如果没做数据库备份，现在就做吧！

13.3.4 EMAIL_BACKEND 和相关的设置

如果你的网站要发送电子邮件，下述设置要正确设置。

默认情况下，Django 发送的邮件使用的发件人是 webmaster@localhost 和 root@localhost，但是有些邮件提供商拒收这两个地址发送的邮件。如果想使用其他发件地址，修改 `DEFAULT_FROM_EMAIL` 和 `SERVER_EMAIL` 设置。

13.3.5 STATIC_ROOT 和 STATIC_URL

开发服务器自动伺服静态文件。但是在生产环境中必须定义 STATIC_ROOT 目录，collectstatic 命令会自动复制那个目录中的静态文件。

13.3.6 MEDIA_ROOT 和 MEDIA_URL

媒体文件由用户上传，是不能信任的！Web 服务器一定不能去解释用户上传的媒体文件。例如，Web 服务器不应该执行用户上传的 .php 文件。现在是检查媒体文件备份策略的好时机。

13.4 HTTPS

允许用户登录的网站都应该强制全站使用 HTTPS，以防明文传输访问令牌。在 Django 中，访问令牌包括用户名、密码、会话 cookie 和密码重设令牌。（如果通过电子邮件发送密码重设令牌，为了保护令牌，你做不了什么。）

保护用户账户或管理后台等敏感区域还不够，因为 HTTP 和 HTTPS 用的是相同的会话 cookie。Web 服务器必须把 HTTP 流量重定向到 HTTPS，只通过 HTTPS 请求 Django。设置好 HTTPS 后，启用下述设置。

13.4.1 CSRF_COOKIE_SECURE

设为 True，以防不小心通过 HTTP 传输 CSRF cookie。

13.4.2 SESSION_COOKIE_SECURE

设为 True，以防不小心通过 HTTP 传输会话 cookie。

13.5 性能优化

设定 DEBUG = False 后，禁用了几个只在开发中有用的功能。此外，可以调整下述几个设置。

13.5.1 CONN_MAX_AGE

如果处理请求时连接数据库所用的时间占据相当一部分，可以启用持久数据库连接，有效减少这一部分的耗时。在网络性能受限的虚拟主机中这么做效果明显。

13.5.2 TEMPLATES

缓存模板加载器通常能极大地提升性能，因为无需每次渲染都编译模板。详情参见模板加载器的文档。

13.6 错误报告

把代码推送到生产环境后，我们都希望能顺利运行，但是意外错误是不可避免的。幸好，Django 能捕获错误，并以合适的方式通知你。

13.6.1 日志

网站上线之前要检查日志配置，而且要确保在有一定流量之后能按预期工作。

13.6.2 ADMINS 和 MANAGERS

出现 500 错误时，会通过邮件通知 ADMINS。出现 404 错误时，会通过邮件通知 MANAGERS。IGNORABLE_404_URLS 可以防止误报。

通过电子邮件报告错误不太便利。在收件箱没被报告淹没之前考虑使用错误监控系统吧，例如 Sentry。Sentry 还能聚合日志。

13.6.3 自定义默认的错误视图

Django 为几个 HTTP 错误码提供了默认的视图和模板。你可以在根模板目录中创建这几个模板覆盖默认的模板：404.html、500.html、403.html 和 400.html。99% 的 Web 应用使用默认视图即可，但是如果你想自定义，请看这里的说明。那份文档还详细说明了下述默认模板：

- http_not_found_view
- http_internal_server_error_view
- http_forbidden_view
- http_bad_request_view

13.7 使用虚拟环境

如果你把项目的 Python 依赖安装到一个虚拟环境中，还要把虚拟环境中的 site-packages 目录添加到 Python 路径中。为此，要把额外的路径添加到 WSGIPythonPath 指令中，在 Unix 类系统中使用冒号（:）分隔多个路径，在 Windows 中则使用分号（;）。如果目录的路径中包含空白字符，整个路径要放在引号内。

```
WSGIPythonPath /path/to/mysite.com:/path/to/your/venv/lib/python3.X/site-packages
```

要提供虚拟环境的正确路径，并把 python3.X 替换成正确的 Python 版本（如 python3.4）。

13.8 在生产环境中使用不同的设置

目前，书中的示例只使用一个设置文件，即 django-admin startproject 命令生成的 settings.py。不过，准备部署时，你可能会发现需要使用多个设置文件，把开发环境和生产环境区分开。（例如，想在本地测试代码时，你可不想每次都把 DEBUG 设置由 False 改成 True。）Django 允许使用多个设置文件，大大简化了这一需求的实现。如果想把生产环境和开发环境的设置分别保存在不同的文件中，可以使用下述三种方法中的一种：

- 使用两个完全独立的设置文件。
- 创建一个基设置文件（比如说针对开发环境），另外创建一个文件（比如说针对生产环境），导入基设置，在里面定义需要覆盖的设置。
- 只使用一个设置文件，通过 Python 逻辑根据上下文修改设置。

下面一一说明。首先，最简单的方法是定义两个独立的设置文件。如果你一直跟着我做，现在已经有了 settings.py 文件。那么，我们只需复制一份，将其命名为 settings_production.py。（文件名是我随便起的，你可以随便命名。）然后，在这个新文件中修改 DEBUG 等设置。第二种方法类似，不过去除了冗余。我们不再创建两个内容差不多的设置文件，而是以一个为基础，将其导入另一个文件。例如：

```
# settings.py
```

```python
DEBUG = True
TEMPLATE_DEBUG = DEBUG

DATABASE_ENGINE = 'postgresql_psycopg2'
DATABASE_NAME = 'devdb'
DATABASE_USER = ''
DATABASE_PASSWORD = ''
DATABASE_PORT = ''

# ...

# settings_production.py

from settings import *

DEBUG = TEMPLATE_DEBUG = False
DATABASE_NAME = 'production'
DATABASE_USER = 'app'
DATABASE_PASSWORD = 'letmein'
```

这里，settings_production.py 从 settings.py 中导入全部设置，然后针对生产环境重新定义一些设置。例如，我们把 DEBUG 设成了 False，还修改了数据库连接参数。（后者是为了说明可以重新定义任何设置，而不是只能重新定义 DEBUG 等基本设置。）

最后，区分不同环境的设置最简洁的方法是只使用一个文件，但是根据条件分别为各个环境定义设置。区分环境的一种方式是检查当前主机名。例如：

```python
# settings.py

import socket

if socket.gethostname() == 'my-laptop':
    DEBUG = TEMPLATE_DEBUG = True
else:
    DEBUG = TEMPLATE_DEBUG = False

# ...
```

这里，我们从 Python 标准库中导入 socket 模块，使用它检查当前系统的主机名。通过检查主机名，可以判断代码是否运行在生产服务器中。这里的关键是，设置文件中的内容就是普通的 Python 代码，可以导入其他文件，可以执行任何逻辑，等等。注意，如果采用这种方法，设置文件中的 Python 代码是可以纠错的。如果抛出异常，Django 有可能崩溃。

settings.py 文件的名称可以随意起，例如 settings_dev.py、settings/dev.py 或 foobar.py，Django 并不在意，只要告诉它你使用的是哪个就行。

但是，如果真的重命名了 django-admin startproject 命令生成的 settings.py，manage.py 会报错，说找不到设置。这是因为，manage.py 尝试导入一个名为 settings 的模块。这个问题的解决方法是，修改 manage.py 文件，把 settings 改为新的模块名，或者用 django-admin 代替 manage.py。如果采用后一种方法，要设定 DJANGO_SETTINGS_MODULE 环境变量，指向设置文件的 Python 路径（例如 'mysite.settings'）。

13.9 把 Django 应用程序部署到生产服务器

别给自己找麻烦

如果你真想部署一个线上网站，明智的做法只有一个：找一个明确支持 Django 的主机。这样的主机不仅自带媒体服务器（通常是 Nginx），还会为你做好一些设置，比如配置好 Apache，以及定期重启 Python 进程的定时任务（cron job，避免网站停机）。有些优秀主机可能还会提供某种形式的“一键”部署功能。

真的，别给自己找麻烦，找个支持 Django 的主机吧，一个月用不了几块钱。

13.10 使用 Apache 和 `mod_wsgi` 部署 Django 应用程序

经证明，使用 Apache 和 `mod_wsgi` 部署 Django 应用程序是有效的方式。`mod_wsgi` 是 Apache 的一个模块，能存贮任何 Python WSGI 应用程序，包括 Django。任何支持 `mod_wsgi` 的 Apache 版本都可以。`mod_wsgi` 的官方文档很全面，详述了 `mod_wsgi` 的方方面面。或许，首先应该阅读安装和配置文档。

13.10.1 基本配置

安装并激活 `mod_wsgi` 之后，编辑 Apache 服务器的 `httpd.conf` 文件，添加下述内容。注意，如果使用的 Apache 版本低于 2.4，把 `Require all granted` 替换成 `Allow from all`，并在前面一行添加 `Order deny,allow`。

```
WSGIScriptAlias / /path/to/mysite.com/mysite/wsgi.py
WSGIPythonPath /path/to/mysite.com

<Directory /path/to/mysite.com/mysite>
<Files wsgi.py>
Require all granted
</Files>
</Directory>
```

`WSGIScriptAlias` 那行里的第一部分是伺服应用程序的基 URL（/ 表示根 URL），第二部分是 WSGI 文件在系统中的位置（参见下文），这个文件通常在项目包中（这里的 `mysite`）。这一行告诉 Apache，使用那个文件中定义的 WSGI 应用程序在指定的 URL 下伺服请求。

`WSGIPythonPath` 那行确保项目包在 Python 路径中，可以导入。也就是说，让 `import mysite` 可用。`<Directory>` 部分确保 Apache 能访问 `wsgi.py` 文件。

接下来要确保 `wsgi.py` 文件中有一个 WSGI 应用程序对象。从 Django 1.4 开始，`startproject` 命令会自动创建；如果使用旧版，需要自行创建。

这个文件中默认的内容，以及可以添加的其他内容参见这篇文档。

> **提醒**
>
> 如果在一个 `mod_wsgi` 进程中运行多个 Django 网站，所有网站都使用第一个运行的设置。如果不想这样，可以把 `wsgi.py` 文件中的
>
> ```
> os.environ.setdefault("DJANGO_SETTINGS_MODULE", "{{ project_name }}.settings")
> ```
>
> 改成：
>
> ```
> os.environ["DJANGO_SETTINGS_MODULE"] = "{{ project_name }}.settings"
> ```
>
> 或者使用 `mod_wsgi` 守护进程模式，并确保各个网站在各自的守护进程中运行。

13.10.2 使用 mod_wsgi 的守护进程模式

推荐使用守护进程模式运行 `mod_wsgi`（Windows 除外）。为了创建所需的守护进程组，并委托它运行 Django 实例，要添加合适的 `WSGIDaemonProcess` 和 `WSGIProcessGroup` 指令。

除此之外，在守护进程模式中不能使用 `WSGIPythonPath`。应该使用 `WSGIDaemonProcess` 指令的 `python-path` 选项，例如：

```
WSGIDaemonProcess example.com python-path=/path/to/mysite.com:/path/to/venv/lib/python2.7/
site-packages
WSGIProcessGroup example.com
```

设置守护进程模式的详细说明参见 `mod_wsgi` 官方文档。

13.11 在生产环境中伺服文件

Django 本身不伺服文件，而是交给你选择的 Web 服务器。建议使用单独的 Web 服务器（即不负责运行 Django 的服务器）伺服媒体文件。下面是两个不错的选择：

- Nginx
- Apache 简化版

然而，如果你别无选择，只能在运行 Django 的 Apache 虚拟主机中伺服媒体文件，可以让 Apache 把某些 URL 视作静态媒体文件，其它 URL 则使用 Django 的 `mod_wsgi` 接口伺服。

下述示例把 Django 放到网站根目录中，但是明确指明把 `robots.txt`、`favicon.ico`、CSS 文件，以及 `/static/` 和 `/media/` 目录中的文件视作静态文件。除此之外的 URL 都使用 `mod_wsgi` 伺服。

```
Alias /robots.txt /path/to/mysite.com/static/robots.txt
Alias /favicon.ico /path/to/mysite.com/static/favicon.ico

Alias /media/ /path/to/mysite.com/media/
Alias /static/ /path/to/mysite.com/static/

<Directory /path/to/mysite.com/static>
Require all granted
</Directory>
```

```
<Directory /path/to/mysite.com/media>
Require all granted
</Directory>

WSGIScriptAlias / /path/to/mysite.com/mysite/wsgi.py

<Directory /path/to/mysite.com/mysite>
<Files wsgi.py>
Require all granted
</Files>
</Directory>
```

如果使用的 Apache 版本低于 2.4，把 `Require all granted` 替换成 `Allow from all`，并在前面一行添加 `Order deny,allow`。

13.11.1 伺服管理后台的文件

如果 `django.contrib.staticfiles` 在 `INSTALLED_APPS` 中，Django 开发服务器自动伺服管理后台（以及安装的其他应用）的静态文件。在其他环境中则不然，为了伺服管理后台的静态文件，你要自行设置 Apache 或你使用的其他 Web 服务器。

管理后台的静态文件在 `django/contrib/admin/static/admin` 中。我们强烈建议使用 `django.contrib.staticfiles` 处理管理后台的静态文件（如前一节所述，也交给 Web 服务器；为此，要使用 `collectstatic` 管理命令收集 `STATIC_ROOT` 中的静态文件，然后配置 Web 服务器，在 `STATIC_URL` 上伺服 `STATIC_ROOT`），不过除此之外还有三种方案：

1. 在文档根目录中创建一个符号链接，指向管理后台的静态文件目录（可能要在 Apache 的配置文件中使用 `+FollowSymLinks`）。
2. 使用 `Alias`（如前所示）为相应的 URL 创建别名（可能是 `STATIC_URL` + `admin/`），指向管理后台静态文件的真正位置。
3. 复制管理后台的静态文件，放到 Apache 的文档根目录中。

13.11.2 如果遇到 `UnicodeEncodeError` 异常

如果你使用 Django 的国际化功能，而且允许用户上传文件，必须确保 Apache 所在的环境接受非 ASCII 文件名。如果没有正确配置环境，使用 `os.path` 中的函数处理包含非 ASCII 字符的文件名时会触发 `UnicodeEncodeError` 异常。

为了避免这类问题，启动 Apache 的环境应该做类似下面的配置：

```
export LANG='en_US.UTF-8'
export LC_ALL='en_US.UTF-8'
```

这些配置项目的具体句法和存放位置参阅操作系统的文档。在 Unix 平台中通常放在 `/etc/apache2/envvars` 目录中。在环境中添加上述配置之后，重启 Apache。

13.12 在生产环境伺服静态文件

在生产环境中伺服静态文件的过程很简单：静态文件有变化时运行 `collectstatic` 命令，然后把静态文件目录（`STATIC_ROOT`）中收集到的文件移到静态文件服务器中伺服。

根据 STATICFILES_STORAGE 设置的不同，你可能要自己动手把静态文件移到新位置，或者让 Storage 类的 post_process 方法代劳。

当然，与所有部署任务一样，细节是最麻烦的。每个生产环境多少都有点区别，因此你要根据自己的需求调整上述基本过程。

下面讨论几种常见做法，希望能给你一些帮助。

13.12.1 使用同一个服务器伺服网站和静态文件

如果想使用伺服网站的服务器伺服静态文件，这个过程可能是这样的：

- 把代码推送到服务器中。
- 在服务器中运行 collectstatic 命令，把所有静态文件复制到 STATIC_ROOT 目录中。
- 配置 Web 服务器，在 STATIC_URL 上伺服 STATIC_ROOT。

你可能想让这个过程实现自动化，尤其是有多个 Web 服务器时。自动化有很多方法，多数 Django 开发者喜欢使用 Fabric。

下面，以及后续几小节将展示几个 fabfile（即 Fabric 脚本）示例，说明如何自动部署静态文件。fabfile 的句法相当简单，不再赘述。如果想全面了解句法，请阅读 Fabric 的文档。把静态文件部署到多个 Web 服务器的 fabfile 可能是下面这样的：

```
from fabric.api import *

# 目标主机
env.hosts = ['www1.example.com', 'www2.example.com']

# 项目代码在服务器中的位置
env.project_root = '/home/www/myproject'

def deploy_static():
    with cd(env.project_root):
        run('./manage.py collectstatic -v0 --noinput')
```

13.12.2 使用专门的服务器伺服静态文件

多数大型 Django 网站使用单独的 Web 服务器（即运行 Django 之外的服务器）伺服静态文件。静态文件专用的服务器通常运行速度更快、但功能不是那么全面的 Web 服务器。常见的选择有：

- Nginx
- Apache 简化版

本书不讲如何配置这些服务器，详细说明参见各服务器的文档。因为静态文件服务器不运行 Django，所以把部署策略修改成下面这样：

1. 静态文件有变化时在本地运行 collectstatic 命令。
2. 把本地的 STATIC_ROOT 推送到静态文件服务器中用于伺服的目录里。这一步通常使用 rsync，因为只需要传输有变化的静态文件。

此时，fabfile 如下所示：

```
from fabric.api import *
from fabric.contrib import project

# 静态文件在本地的位置，即 STATIC_ROOT 设置
env.local_static_root = '/tmp/static'

# 静态文件在远程服务器中的位置
env.remote_static_root = '/home/www/static.example.com'

@roles('static')
def deploy_static():
    local('./manage.py collectstatic')
    project.rsync_project(
        remote_dir = env.remote_static_root,
        local_dir = env.local_static_root,
        delete = True
    )
```

13.12.3 使用云服务或 CDN 伺服静态文件

另一种常见的策略是使用云存储提供商（如 Amazon 的 S3）和（或）CDN（Content Delivery Network，内容分发网络）伺服静态文件。采用这种方式，你无须费心伺服静态文件，而且网页加载的速度通常更快（使用 CDN 时更是如此）。

使用这些服务时，基本的工作流程与前面很像，但是不再使用 `rsync` 传输静态文件，而是要自己把静态文件上传到存储提供商或 CDN 中。这一步有很多做法，不过如果提供商有 API 的话，可以自定义文件存储后端，极大地简化这个过程。

如果你自己编写了存储后端，或者使用第三方存储后端，可以把 `STATICFILES_STORAGE` 设为自定义的存储引擎，供 `collectstatic` 使用。假如你写的 S3 存储后端在 `myproject.storage.S3Storage` 中，可以这么设置：

```
STATICFILES_STORAGE = 'myproject.storage.S3Storage'
```

设置好之后，只需运行 `collectstatic` 命令，这样静态文件就会通过指定的存储后端推送到 S3 中。如果以后想换成其他存储提供商，只需修改 `STATICFILES_STORAGE` 设置。很多常用的文件存储 API 都有第三方应用提供存储后端。你可以到 Django Packages 中找一找。

13.13 弹性伸缩

现在，我们知道如何在单个服务器中运行 Django 了。下面说明如何弹性伸缩。本节说明如何把网站部署到大规模集群中，以便每小时伺服百万级访问。然而，要知道， 各个大型网站之间是有区别的，因此没有完全通用的做法。

接下来的内容说明常规的原则，以及何时可以选用其他方案。首先，我们专讲使用 Apache 和 `mod_python` 时如何弹性伸缩——这是大前提。虽然有很多中大型网站使用 FastCGI，但是我们对 Apache 更熟悉。

13.13.1 在单个服务器中运行

多数网站一开始在单个服务器中运行，采用的架构类似于图 13-1。然而，随着流量的增加，软件的不同部分之间很快就会出现资源竞争（resource contention）。

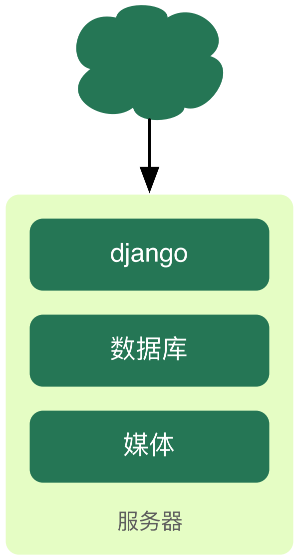


图 13-1：Django 在单个服务器中的架构

数据库服务器和 Web 服务器都喜欢独占整个服务器，如果在同一个服务器中运行，它们会争用资源（RAM、CPU），都想占为己有。把数据库服务器移到单独的设备中就能轻易解决这个问题。

13.13.2 把数据库服务器独立出来

对 Django 来说，把数据库服务器独立出来的过程极其简单，只需把 `DATABASE_HOST` 设置改为数据库服务器的 IP 或 DNS。如果可能，使用 IP 最好，不推荐使用 DNS 连接 Web 服务器和数据库服务器。数据库服务器独立后，架构如图 13-2 所示。



图 13-2：把数据库移到专门的服务器中

这就是我们经常说的 n 层（n-tier）架构。别被这个术语吓到了，它的意思就是把 Web 栈的不同层分到不同的物理设备中。

此时，如果预期将使用多个数据库服务器，或许应该开始考虑连接池和（或）数据库复制（database replication）。可惜，本书没有足够的篇幅探讨这个话题，如果你想进一步了解，可以阅读数据库的文档，或者寻求社区的帮助。

13.13.3 使用单独的服务器伺服媒体文件

单个服务器还有一个大问题没有考虑到：媒体文件和动态内容在同一个服务器中处理。这两方面若想都获得最好的性能，需要不同的环境，如果挤在一起，性能上可谓两败俱伤。

因此，还要把媒体文件独立出去，即把不是 Django 视图生成的内容移到专门的服务器中（见图 13-3）。

理想情况下，媒体服务器最好使用简化版 Web 服务器，并且要做优化，能直接伺服静态媒体文件。这里，Nginx 是最好的选择，不过使用 lighttpd 或极度简化的 Apache 也行。对静态内容（照片、视频，等等）较多的网站来说，单独使用一个媒体文件服务器更加重要，而且在弹性伸缩时，应该放在第一步。

不过，这个过程可能很棘手。如果应用程序允许上传文件，Django 要能够把上传的媒体文件写入媒体服务器。如果媒体文件存贮在另一个服务器中，要提供一种方式，以便通过网络写数据。

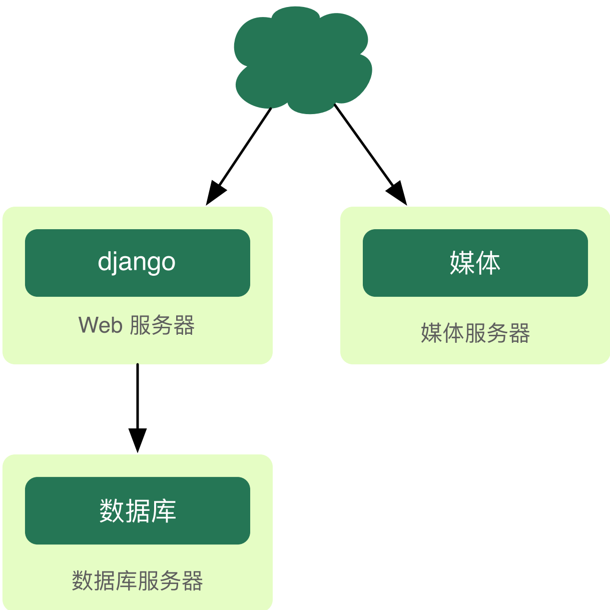


图 13-3：把媒体服务器独立出去

13.13.4 实现负载均衡和冗余

目前，我们尽可能做了分解。这种三服务器架构应该能处理相当多的流量了（我们的一个网站使用这种架构，每天能伺服一千万左右的访问量），如果网站进一步扩容，需要添加冗余了。

冗余确实是个好东西。看一眼图 13-3，你会发现，即便三个服务器中有一个宕机了，整个网站都将无法访问。因此，添加冗余服务器不仅增加了容量，还提升了可靠性。这里，我们暂且假设 Web 服务器首先达到容量瓶颈。

在不同的硬件中运行多份 Django 网站比较简单，只需把代码复制到多台设备中，然后在各台设备中启动

Apache。不过，你还需要一种软件，通过它把流量分发给多台服务器，即负载均衡程序（load balancer）。

你可以花钱购买专用的硬件负载均衡程序，不过也有一些高质量的开源负载均衡程序可用。Apache 的 `mod_proxy` 就是其中一个，不过笔者觉得 Perlbal 更好。Perlbal 是一个负载均衡程序和反向代理，由 `memcached`（参见第 16 章）的开发者们编写。

We 服务器集群化之后，得到的结构更为复杂，如图 13-4 所示。

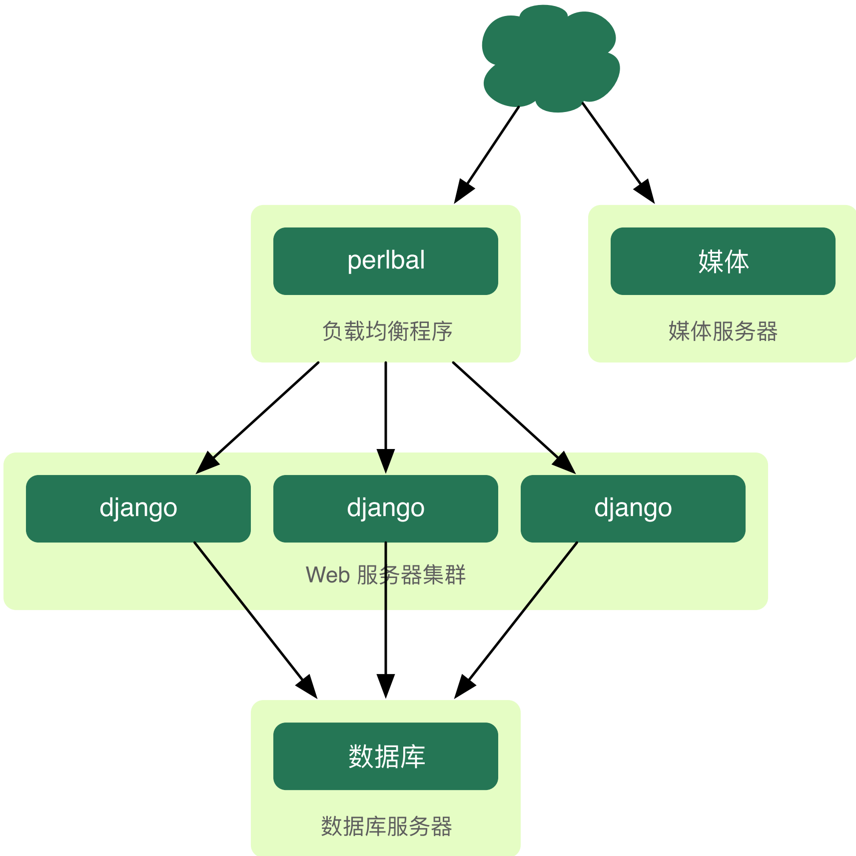


图 13-4：负载均衡的冗余服务器架构

注意，图中把多台 Web 服务器称为一个集群，以此表明 Web 服务器的数量基本上是可变的。在前面加上负载均衡程序之后，可以轻易添加和删除后端 Web 服务器，一秒也不用停机。

13.13.5 发展壮大

接下来，可以考虑以下几件事：

- 如果想进一步提升数据库的性能，可以复制数据库服务器。MySQL 内建支持复制；PostgreSQL 用户可以使用 Slony 复制数据库，使用 pgpool 管理连接池。
- 如果一个负载均衡程序不够用，可以在前面添加多个，然后使用循环（round-robin）DNS 在各个负载均衡程序之间分发。
- 如果一个媒体服务器不够用，可以添加多个，然后使用负载均衡集群分发负载。
- 如果需要更多的缓存存储器容量，可以添加专门的缓存服务器。
- 任何时候，只要觉得集群的性能不够，就可以在集群中添加更多服务器。

这样调整之后，大规模集群架构类似于图 13-5。

虽然图中在每一层只给出了两三个服务器，但是你可以添加更多，每一层基本上没有数量限制。

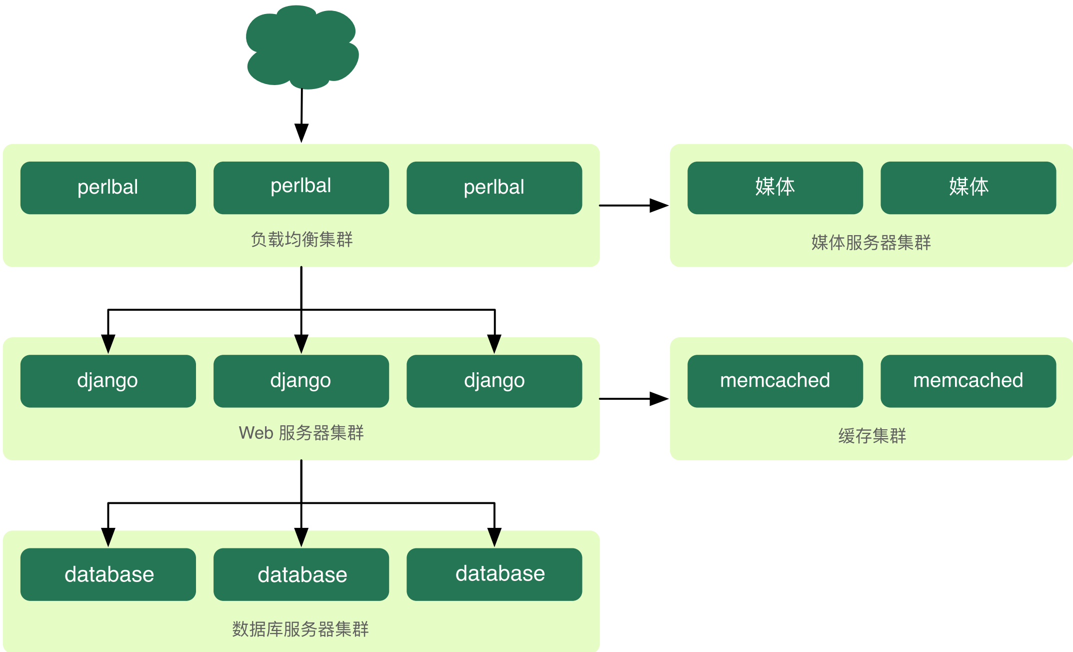


图 13-5：**大规模 Django 架构示例**

13.14 性能调优

如果对你来说钱不是问题，弹性伸缩时只需投资更多的硬件。但是，一般来说，我们的资金是有限的，因此必须调优性能。

> **提示**
>
> 如果阅读本书的你碰巧手握巨资，请考虑给 Django 基金会捐一笔。他们也接受未经切割的钻石和金元宝。

可是，性能调优是一门艺术而非科学，与弹性伸缩相比，更难道出个所以然。如果你真想部署大型 Django 应用程序，必须花大量时间学习如何调优各个部分。

不过，接下来的几小节将讨论这些年笔者发现的一些对 Django 来说行之有效的调优技巧。

13.14.1 RAM 不嫌多

曾经昂贵的 RAM 如今我们都能负担得起了。尽你所能，尽量多购买 RAM，越多越好。处理器再快也提升不了多少性能，对多数 Web 服务器来说，90% 的时间都在等待磁盘 I/O。一旦开始使用交换内存，性能便会急剧下降。速度更快的磁盘可能有点帮助，但是价格比 RAM 贵，因此没必要在磁盘上投资太多。

如果有多个服务器，首先应该为数据库服务器增加 RAM。如果你能负担得起，多加一些 RAM，把整个数据库都载入内存。这没什么问题，笔者开发的一个网站有 50 多万篇新闻稿，而空间只占了 2GB。

其次，尽量为 Web 服务器增加 RAM。理想的情况是绝不使用交换内存。如果达到这个水平，你就能应付大

多数正常的流量。

13.14.2 禁用 Keep-Alive

`Keep-Alive` 是 HTTP 的一个特性，目的是让多个 HTTP 请求通过一次 TCP 连接伺服，从而避免接断 TCP 时的消耗。乍一看，这是个好主意，但是 Django 网站的性能可能受到影响。如果使用单独的服务器伺服媒体文件，用户浏览你的网站时，差不多每十秒才会从 Django 服务器中请求一个页面。这样，HTTP 服务器要一直等待下一个请求，而空闲的 HTTP 服务器与工作时消耗的 RAM 一样多。

13.14.3 使用 memcached

尽管 Django 支持不同的缓存后端，但是它们都没有 memcached 速度快。如果你的网站流量很大，别考虑其他后端，使用 memcached 就对了。

13.14.4 经常使用 memcached

当然，如果不用，memcached 就是个摆设。第 16 章将详细说明如何使用 Django 的缓存框架，以及如何尽量多地使用。在大流量下，往往只有优先抢占式缓存才能保住网站不宕机。

13.14.5 参与其中

Django 栈的每一个组成部分，从 Linux 到 Apache，到 PostgreSQL 或 MySQL，背后都有非常棒的社区。如果你真想榨干服务器的性能，参与软件背后的开源社区，向社区中的人寻求帮助。免费软件社区的多数成员都乐于助人。别忘了加入 Django 社区，这里聚集着一群极具活力的 Django 开发者，一直在发展壮大，有大量经验等着传授给你。

13.15 接下来

余下的几章讨论你可能用得到也可能用不到的 Django 功能，你可以根据需要以任何顺序阅读。

第 14 章 生成非 HTML 内容

当我们讨论开发网站时，通常指的是生成 HTML。当然，除了 HTML，Web 中还有很多其他内容。我们通过 Web 分发各种格式的数据，例如 RSS、PDF、图像，等等。

目前，我们讨论的是常规情况，即生成 HTML，不过本章将换个话题，说明如何使用 Django 生成其他类型的内容。Django 为下列常用的非 HTML 内容提供了内置工具：

- 逗号分隔的文件（CSV），便于导入电子表格应用程序
- PDF 文件
- RSS/Atom 订阅源
- 网站地图（一种 XML 格式的文件，最初由 Google 开发，作用是为搜索引擎指路）

后文会分述这些工具，不过在此之前我们先来了解一些基本的原则。

14.1 基础知识：视图和 MIME 类型

第 2 章说过，视图函数就是普通的 Python 函数，它接受一个 Web 请求为参数，返回一个 Web 响应。响应可以是网页的 HTML 内容、重定向、404 错误、XML 文档、图像，等等。更为正式地说，Django 视图必须满足下述两个条件：

1. 第一个参数为一个 `HttpRequest` 实例
2. 返回一个 `HttpResponse` 实例

让视图返回非 HTML 内容的关键在于 `HttpResponse` 类，尤其是 `content_type` 参数。Django 默认把 `content_type` 的值设为 `"text/html"`。不过，你可以把它设为官方的媒体类型（MIME 类型，由 IANA 管理）中的任何一个值。

调整 MIME 类型的目的是告诉浏览器，返回的响应是其他格式。例如，下述视图返回一张 PNG 图像。简单起见，我们直接从磁盘中读取图像文件。

```
from django.http import HttpResponse

def my_image(request):
    image_data = open("/path/to/my/image.png", "rb").read()
    return HttpResponse(image_data, content_type="image/png")
```

就这么简单！如果你把 `open()` 调用中的图像路径换成真实的路径，通过这个简单的视图就能伺服一张图像，浏览器能正确把它显示出来。

还要知道的一点是，`HttpResponse` 对象实现了 Python 的标准文件类对象 API。这意味着，在 Python（或第三方库）期待文件的地方都可以使用 `HttpResponse` 实例。下面通过示例说明如何让 Django 生成 CSV 文件。

14.2 生成 CSV 文件

Python 自带了一个 CSV 库，csv。在 Django 中之所以能使用它创建 CSV 文件，是因为 csv 模块操作的是类似文件的对象，而 Django 的 HttpResponse 对象就是类似文件的对象。下面举个例子：

```
import csv
from django.http import HttpResponse

def some_view(request):
    # 使用恰当的 CSV 首部创建 HttpResponse 对象
    response = HttpResponse(content_type='text/csv')
    response['Content-Disposition'] = 'attachment;
      filename="somefilename.csv"'

    writer = csv.writer(response)
    writer.writerow(['First row', 'Foo', 'Bar', 'Baz'])
    writer.writerow(['Second row', 'A', 'B', 'C', '"Testing"'])

    return response
```

这段代码和注释不解自明，不过有些地方需要注意：

- 响应的 MIME 类型是 text/csv。它的目的是告诉浏览器，这是一个 CSV 文件，而不是 HTML 文件。如果不这样设定，浏览器可能会把输出解释为 HTML，在浏览器窗口中显示杂乱无章的内容。
- 还为响应设定了 Content-Disposition 首部，其值包含 CSV 文件的名称。文件名可以根据需要随意起。浏览器调用另存为对话框时会使用这个名称。
- 调用生成 CSV 的 API 很简单，只需把 response 作为第一个参数传给 csv.writer 函数。 这个函数期待参数是一个类似文件的对象，而 HttpResponse 对象正是这样的对象。
- CSV 文件中的每一行通过一次 writer.writerow 调用写入，传给它的参数是一个可迭代对象，例如列表或元组。
- csv 模块会代为添加引号，因此你不用担心转义包含引号或逗号的字符串。只需把原始字符串传给 writerow() 函数，它能正确处理。

14.2.1 以流的形式发送大 CSV 文件

如果视图生成特别大的响应，可能要考虑使用 Django 的 StreamingHttpResponse。例如，如果生成文件所用的时间很长，负载均衡程序可能会切断连接，以防生成响应时超时；以流的形式发送文件则能避免这种问题。在下述示例中，我们充分利用 Python 生成器高效处理大型 CSV 文件的生成和传输：

```
import csv

from django.utils.six.moves import range
from django.http import StreamingHttpResponse

class Echo(object):
    """An object that implements just the write method of the file-like
    interface.
    """
    def write(self, value):
        """Write the value by returning it, instead of storing in a buffer."""
```

```
        return value

def some_streaming_csv_view(request):
    """A view that streams a large CSV file."""
    # 生成行序列
    # 范围根据多数电子表格应用程序在单个表中能处理的行数而定
    rows = (["Row {}".format(idx), str(idx)] for idx in range(65536))
    pseudo_buffer = Echo()
    writer = csv.writer(pseudo_buffer)
    response = StreamingHttpResponse((writer.writerow(row)
      for row in rows), content_type="text/csv")
    response['Content-Disposition'] = 'attachment;
      filename="somefilename.csv"'
    return response
```

14.2.2 使用模板系统

此外，还可以使用 Django 的模板系统生成 CSV。这比使用 Python 的 csv 模块低端一些，不过为了全面讲解不同方式，特在此说明。基本思想是，把项目列表传给模板，让模板在一个 for 循环中输出逗号。下述示例生成的 CSV 文件与前面一样：

```
from django.http import HttpResponse
from django.template import loader, Context

def some_view(request):
    # 使用恰当的 CSV 首部创建 HttpResponse 对象
    response = HttpResponse(content_type='text/csv')
    response['Content-Disposition'] = 'attachment;
      filename="somefilename.csv"'

    # 这里，数据是硬编码的
    # 还可以从数据库或其他源中加载
    csv_data = (
        ('First row', 'Foo', 'Bar', 'Baz'),
        ('Second row', 'A', 'B', 'C', '"Testing"', "Here's a quote"),
    )

    t = loader.get_template('my_template_name.txt')
    c = Context({'data': csv_data,})
    response.write(t.render(c))
    return response
```

与前面的示例唯一的不同是，这个示例使用模板，而不是 csv 模块。其余的代码，如 content_type='text/csv'，都是一样的。然后，创建 my_template_name.txt 模板，写入下述代码：

```
{% for row in data %}
    "{{ row.0|addslashes }}",
    "{{ row.1|addslashes }}",
    "{{ row.2|addslashes }}",
    "{{ row.3|addslashes }}",
    "{{ row.4|addslashes }}"
{% endfor %}
```

这个模板十分简单，只是迭代给定的数据，逐行显示 CSV 文件的内容。这里使用 `addslashes` 模板过滤器是为了确保添加引号时不出错。

14.3 其他基于文本的格式

注意，CSV 没什么特别的，只是输出的格式特殊。你可以使用上述方式输出任何基于文本的格式。甚至还可以使用类似的方式输出二进制数据，例如 PDF。

14.4 生成 PDF 文件

Django 能使用视图动态输出 PDF 文件。在这个过程中要使用优秀的开源 Python PDF 库 ReportLab。动态生成 PDF 文件的好处是，可以基于不同的目的创建所需的 PDF 文件，例如为不同的用户生成不同的内容。

14.4.1 安装 ReportLab

PyPI 中有 ReportLab 库。用户指南（恰巧是一个 PDF 文件）也可下载。可以使用 `pip` 安装 ReportLab：

```
$ pip install reportlab
```

然后，在 Python 交互式解释器中导入，测试是否成功安装：

```
>>> import reportlab
```

如果这个命令不抛出错误，说明成功安装了。

14.4.2 编写视图

在 Django 中能使用 ReportLab 动态生成 PDF 的关键之处在于，与 `csv` 库一样，ReportLab API 操作的也是类似文件的对象，例如 Django 的 `HttpResponse`。下面是一个 Hello World 示例：

```
from reportlab.pdfgen import canvas
from django.http import HttpResponse

def some_view(request):
    # 使用恰当的 PDF 首部创建 HttpResponse 对象
    response = HttpResponse(content_type='application/pdf')
    response['Content-Disposition'] = 'attachment;
      filename="somefilename.pdf"'

    # 创建 PDF 对象，把 response 对象当做“文件”
    p = canvas.Canvas(response)

    # 在 PDF 对象上绘制内容，即生成 PDF
    # 全部功能参见 ReportLab 的文档
    p.drawString(100, 100, "Hello world.")

    # 关闭 PDF 对象，然后收工
    p.showPage()
    p.save()
    return response
```

这段代码和注释不解自明，不过有些地方需要注意：

- 响应的 MIME 类型是 application/pdf。它的目的是告诉浏览器，这是一个 PDF 文件，而不是 HTML 文件。
- 还为响应设定了 Content-Disposition 首部，其值包含 PDF 文件的名称。文件名可以根据需要随意起。浏览器调用另存为对话框时会使用这个名称。 这里，Content-Disposition 首部开头的值是 'attachment; '。这么做的目的是，即便设备中设置了默认使用哪个程序打开文件，仍然弹出一个对话框，让用户决定怎么做。如果没有 'attachment;'，浏览器会使用为 PDF 配置的程序或插件打开 PDF 文件。后一种情况的代码如下：

  ```
  response['Content-Disposition'] = 'filename="somefilename.pdf"'
  ```

- 调用 ReportLab API 很简单，只需把 response 作为第一个参数传给 canvas.Canvas。Canvas 类期待参数是一个类似文件的对象，而 HttpResponse 对象正是这样的对象。
- 注意，后续生成 PDF 的方法都在 PDF 对象（这里的 p）上调用，而不在 response 上调用。
- 最后，记得要在 PDF 对象上调用 showPage() 和 save() 方法。

14.4.3 复杂的 PDF 文件

使用 ReportLab 创建复杂的 PDF 文件时，应该考虑使用 io 库临时储存 PDF 文件。这个库提供类似文件对象的接口，效率特别高。下面是使用 io 库重写的 Hello World 示例：

```
from io import BytesIO
from reportlab.pdfgen import canvas
from django.http import HttpResponse

def some_view(request):
    # 使用恰当的 PDF 首部创建 HttpResponse 对象
    response = HttpResponse(content_type='application/pdf')
    response['Content-Disposition'] = 'attachment;
      filename="somefilename.pdf"'

    buffer = BytesIO()

    # 创建 PDF 对象，把 BytesIO 对象当做“文件”
    p = canvas.Canvas(buffer)

    # 在 PDF 对象上绘制内容，即生成 PDF
    # 全部功能参见 ReportLab 的文档
    p.drawString(100, 100, "Hello world.")

    # 关闭 PDF 对象
    p.showPage()
    p.save()

    # 获取 BytesIO 缓冲的值，写入响应
    pdf = buffer.getvalue()
    buffer.close()
    response.write(pdf)
    return response
```

14.4.4 其他资源

- PDFlib 也是有 Python 绑定的 PDF 生成库。在 Django 中使用的方式与前文一样。
- Pisa XHTML2PDF 也是一个 PDF 生成库，而且提供了集成到 Django 中的示例。
- HTMLdoc 是一个命令行脚本，用于把 HTML 转换成 PDF。它没有 Python 接口，不过可以采用迂回战术，先使用 `system` 或 `popen` 调用 shell 命令，再使用 Python 取回输出。

14.5 其他可能

使用 Python 能生成众多不同类型的内容。下面简略说明不同类型的生成方式，以及所需的库：

- ZIP 文件：Python 标准库中的 `zipfile` 模块能读写压缩的 ZIP 文件。你可以使用这个模块按需归档一堆文件，或者压缩大文档。类似地，可以使用标准库中的 `tarfile` 模块生成 TAR 文件。
- 动态生成图像：Python Imaging Library（PIL）是个优秀的库，用于生成图像（PNG、JPEG、GIF，等等）。可以使用这个库自动缩放图像，生成缩略图，把多个图像合并到一个图框中，甚至还能在 Web 应用程序中处理图像。
- 各种图表：有些强大的 Python 图表库可以按需生成地图和图表等图形。这方面的库太多了，无法一一列出，下面给出两个值得一用的：
    - matplotlib 可用于生成通常由 MatLab 或 Mathematica 才能生成的高质量散点图。
    - pygraphviz 是 Graphviz 图形布局工具包的接口，可用于生成图形和网络的结构图。

一般来说，能够写入文件的 Python 库都能在 Django 中使用，因此有无限可能。

至此，我们简要说明了如何生成非 HTML 内容，下面上升一个层次，讲讲抽象方面。Django 为一些常用的非 HTML 内容提供了十分便利的生成工具。

14.6 订阅源框架

Django 自带了一个抽象的订阅源生成框架，可以轻易创建 RSS 和 Atom 订阅源。RSS 和 Atom 都是基于 XML 的格式，用于自动提供网站内容的更新源。RSS 的详情参见这里，Atom 的详情参见这里。

若想创建订阅源，只需编写一个简短的 Python 类。订阅源的数量不限。Django 还为生成订阅源提供了低层 API。如果想在 Web 之外的场合或者以其他低层的方式生成订阅源，可以使用这个 API。

14.6.1 抽象框架

概述

订阅源生成抽象框架由 `Feed` 类提供。若想创建一个订阅源，编写一个 `Feed` 的子类，然后在 URL 配置中指向它的实例。

订阅源类

订阅源类是表示订阅源的 Python 类。订阅源可以简单（例如网站中的新闻订阅源，或者显示最新博客文章的订阅源），也可以复杂（例如显示某一分类中所有博客文章的订阅源，其中分类是可变的）。订阅源类是 `django.contrib.syndication.views.Feed` 的子类，可以放在代码基中的任何位置。订阅源类的实例是可以在 URL 配置中使用的视图。

一个简单的示例

这个简单的示例摘自虚构的警务新闻网站，用于显示最新的五篇新闻稿：

```
from django.contrib.syndication.views import Feed
from django.core.urlresolvers import reverse
from policebeat.models import NewsItem

class LatestEntriesFeed(Feed):
    title = "Police beat site news"
    link = "/sitenews/"
    description = "Updates on changes and additions to police beat central."

    def items(self):
        return NewsItem.objects.order_by('-pub_date')[:5]

    def item_title(self, item):
        return item.title

    def item_description(self, item):
        return item.description

    # 仅当 NewsItem 未定义 get_absolute_url 方法时才需要 item_link
    def item_link(self, item):
        return reverse('news-item', args=[item.pk])
```

为了在一个 URL 上开放这个订阅源，要在 URL 配置中添加这个类的一个对象。例如：

```
from django.conf.urls import url
from myproject.feeds import LatestEntriesFeed

urlpatterns = [
    # ...
    url(r'^latest/feed/$', LatestEntriesFeed()),
    # ...
]
```

注意：

- 订阅源类是 `django.contrib.syndication.views.Feed` 的子类。
- `title`、`link` 和 `description` 属性分别对应于 RSS 标准中的 `<title>`、`<link>` 和 `<description>` 元素。
- `items()` 方法的作用很简单，返回一个对象列表，在订阅源中以 `<item>` 元素呈现。虽然这个示例使用 Django 的对象关系映射器返回一组 `NewsItem` 对象，但是不一定非得返回模型实例。`items()` 可以返回任何类型的对象，不过返回 Django 模型更便利。
- 如果创建的是 Atom 订阅源，而不是 RSS 订阅源，别设定 `description` 属性，应该设定 `subtitle` 属性。“同时提供 Atom 和 RSS 订阅源”一节有个例子。

还有一件事没做。在 RSS 订阅源中，每个 `<item>` 中都有 `<title>`、`<link>` 和 `<description>`。我们要告诉框架这些元素中要放什么数据。

为了生成 `<title>` 和 `<description>` 元素的内容，Django 尝试调用订阅源类的 `item_title()` 和 `item_description()` 方法。这两个方法的参数都是 `item`，即对象本身。这两个方法可以不定义，Django 默认使用对象的

Unicode 表示形式。

如果想在标题或描述中使用特殊的格式，可以使用 Django 模板。模板的路径由订阅源类的 title_template 和 description_template 属性指定。这两个模板在渲染各个条目时调用，而且有两个模板上下文变量：

- {{ obj }}：当前对象（items() 返回的对象之一）。
- {{ site }}：Django 中表示当前网站的 site 对象。可以获取 {{ site.domain }} 或 {{ site.name }}。

“一个复杂的示例”一节将说明如何使用模板渲染描述。

除了上述两个模板变量之外，如果还想为标题和描述模板传递额外信息，也有办法：在 Feed 的子类中实现 get_context_data 方法。例如：

```
from mysite.models import Article
from django.contrib.syndication.views import Feed

class ArticlesFeed(Feed):
    title = "My articles"
    description_template = "feeds/articles.html"

    def items(self):
        return Article.objects.order_by('-pub_date')[:5]

    def get_context_data(self, **kwargs):
        context = super(ArticlesFeed, self).get_context_data(**kwargs)
        context['foo'] = 'bar'
        return context
```

模板：

```
Something about {{ foo }}: {{ obj.description }}
```

这个方法在 items() 返回的各个条目上调用，传入下述关键字参数：

- item：当前条目。为了向后兼容，这个上下文变量的名称是 {{ obj }}。
- obj：get_object() 返回的对象。为了避免与 {{ obj }}（参见上一条）混淆，默认没有把这个变量开放给模板，但是可以在你实现的 get_context_data() 方法中使用。
- site：如前所述，是当前网站。
- request：当前请求。

get_context_data() 的行为类似于通用视图，要调用 super() 从父类中获取上下文数据，添加数据后再返回修改后的字典。

指定 <link> 元素的内容有两种方式。对 items() 返回的各个条目，Django 首先尝试调用订阅源类的 item_link() 方法。与标题和描述类似，它的参数也是 item。如果这个方法不存在，Django 尝试在条目上调用 get_absolute_url()。

get_absolute_url() 和 item_link() 都应该以常规 Python 字符串的形式返回条目的 URL。与 get_absolute_url() 一样，item_link() 的结果也直接作为 URL 使用，因此你要负责对 URL 做必要的处理，例如添加引号、转换成 ASCII。

一个复杂的示例

这个框架也支持通过参数实现的复杂订阅源。例如，网站可能会为城市中每个警员最近破获的犯罪活动提供 RSS 订阅源。如果为每个警员都单独创建一个订阅源类，那就太荒唐了。若是真这么做，违背了 DRY 原则不说，还耦合了数据和程序设计逻辑。

其实，订阅源框架支持访问 URL 配置传过来的参数，因此订阅源可以根据 URL 中的信息输出相关条目。各警员的订阅源可以通过这样的 URL 访问：

- `/beats/613/rss/`：返回 613 号警员最近破获的犯罪活动。
- `/beats/1424/rss/`：返回 1424 号警员最近破获的犯罪活动。

这种 URL 在 URL 配置中可以这样匹配：

```
url(r'^beats/(?P[0-9]+)/rss/$', BeatFeed()),
```

与视图类似，这里的 URL 参数连同请求对象一起传给 `get_object()` 方法。各警员的订阅源可以这样实现：

```
from django.contrib.syndication.views import FeedDoesNotExist
from django.shortcuts import get_object_or_404

class BeatFeed(Feed):
    description_template = 'feeds/beat_description.html'

    def get_object(self, request, beat_id):
        return get_object_or_404(Beat, pk=beat_id)

    def title(self, obj):
        return "Police beat central: Crimes for beat %s" % obj.beat

    def link(self, obj):
        return obj.get_absolute_url()

    def description(self, obj):
        return "Crimes recently reported in police beat %s" % obj.beat

    def items(self, obj):
        return Crime.objects.filter(beat=obj).order_by('-crime_date')[:30]
```

Django 使用 `title()`、`link()` 和 `description()` 方法生成订阅源中的 `<title>`、`<link>` 和 `<description>` 元素。

在前面的示例中，这三个元素的内容使用类属性指定，而这个示例表明，除此之外还可以使用方法。生成这三个元素时，Django 按下述顺序操作：

1. 首先，调用相应的方法，传入 `obj` 参数。这里的 `obj` 是 `get_object()` 返回的对象。
2. 如果失败，尝试调用相应的方法时不传入参数。
3. 如果失败，使用相应的类属性。

注意，`items()` 方法也是如此：首先尝试 `items(obj)`，然后尝试 `items()`，最后使用 `items` 类属性（其值为一个列表）。这里，我们使用模板生成条目的描述。这个模板的内容可以非常简单：

```
{{ obj.description }}
```

不过，你可以根据需要添加格式。下文的 ExampleFeed 类将详细说明订阅源类的各个方法和属性。

指定订阅源的类型

这个框架生成的订阅源默认使用 RSS 2.0 格式。若想更改，在订阅源类中添加 feed_type 属性，如下所示：

```
from django.utils.feedgenerator import Atom1Feed

class MyFeed(Feed):
    feed_type = Atom1Feed
```

注意，feed_type 的值是一个类对象，而不是实例。目前可用的订阅源类型有：

- django.utils.feedgenerator.Rss201rev2Feed（RSS 2.01，默认值）
- django.utils.feedgenerator.RssUserland091Feed（RSS 0.91）
- django.utils.feedgenerator.Atom1Feed（Atom 1.0）

附件

若想指定附件，例如播客订阅源中的音频文件，使用 item_enclosure_url、item_enclosure_length 和 item_enclosure_mime_type 钩子。用法举例参见下文的 ExampleFeed 类。

语言

这个框架创建的订阅源会自动添加适当的 <language> 标签（RSS 2.0）或 xml:lang 属性（Atom）。值直接取自 LANGUAGE_CODE 设置。

URL

link 方法（属性）可以返回一个绝对路径（如 /blog/）或包含域名和协议的完整 URL（如 http://www.example.com/blog/）。如果 link 的值不包含域名，订阅源框架会根据 SITE_ID 设置插入当前网站的域名。Atom 订阅源要求有个 <link rel="self">，定义订阅源的当前位置。订阅源框架会根据 SITE_ID 设置，自动把这个元素的值设为当前网站的域名。

同时提供 Atom 和 RSS 订阅源

有些开发者喜欢同时提供 Atom 和 RSS 订阅源。在 Django 中这很容易实现：首先编写一个订阅源类的子类，把 feed_type 属性设为相应的值，然后在 URL 配置中添加另一个版本。下面是一个完整的示例：

```
from django.contrib.syndication.views import Feed
from policebeat.models import NewsItem
from django.utils.feedgenerator import Atom1Feed

class RssSiteNewsFeed(Feed):
    title = "Police beat site news"
    link = "/sitenews/"
    description = "Updates on changes and additions to police beat central."

    def items(self):
        return NewsItem.objects.order_by('-pub_date')[:5]

class AtomSiteNewsFeed(RssSiteNewsFeed):
```

```
    feed_type = Atom1Feed
    subtitle = RssSiteNewsFeed.description
```

提示

在这个示例中，RSS 订阅源使用的是 description，而 Atom 订阅源使用的是 subtitle。这是因为 Atom 订阅源不提供整个订阅源的描述，而提供子标题。如果在订阅源类中设定了 description，Django 不会自动将其用作 subtitle，因为这二者不一定是相同的。在 Atom 订阅源中就应该定义 subtitle 属性。

在上述示例中，我们直接把 Atom 订阅源的 subtitle 设为 RSS 订阅源的 description，因为描述已经很简短了。相应的 URL 配置如下：

```
from django.conf.urls import url
from myproject.feeds import RssSiteNewsFeed, AtomSiteNewsFeed

urlpatterns = [
    # ...
    url(r'^sitenews/rss/$', RssSiteNewsFeed()),
    url(r'^sitenews/atom/$', AtomSiteNewsFeed()),
    # ...
]
```

提示

订阅源类可用的全部属性和方法参见文档中的示例。

14.6.2 低层 API

抽象的订阅源框架在背后使用低层 API 生成订阅源的 XML。低层 API 全在 django/utils/feedgenerator.py 一个模块中。我们可以使用这些 API 自己动手生成订阅源，也可以针对其他的订阅源类型自定义生成订阅源的子类。

SyndicationFeed 类及其子类

feedgenerator 模块中有个基类：

- django.utils.feedgenerator.SyndicationFeed

以及它的几个子类：

- django.utils.feedgenerator.RssUserland091Feed
- django.utils.feedgenerator.Rss201rev2Feed
- django.utils.feedgenerator.Atom1Feed

这三个类都知道如何渲染相应类型的订阅源 XML，它们具有相同的接口：

`SyndicationFeed.__init__()`

使用包含元数据（应用于整个订阅源）的字典初始化订阅源。必须的关键字参数有：

- `title`
- `link`
- `description`

此外还有很多可选的关键字参数：

- `language`
- `author_email`
- `author_name`
- `author_link`
- `subtitle`
- `categories`
- `feed_url`
- `feed_copyright`
- `feed_guid`
- `ttl`

传给 `__init__` 方法的其他关键字参数都存储在 `self.feed` 中，可在自定义的订阅源生成器中使用。所有参数的值都应该是 Unicode 对象，但是 `categories` 例外，它的值应该是 Unicode 对象序列。

`SyndicationFeed.add_item()`

使用指定的参数创建一个条目，添加到订阅源中。

必须的关键字参数有：

- `title`
- `link`
- `description`

可选的关键字参数有：

- `author_email`
- `author_name`
- `author_link`
- `pubdate`
- `comments`
- `unique_id`
- `enclosure`
- `categories`

- `item_copyright`
- `ttl`
- `updateddate`

其他关键字参数也会存储下来，以便在自定义的订阅源生成器中使用。如果指定，除了下述参数之外，所有参数的值都应该是 Unicode 对象：

- `pubdate` 的值应该是一个 Python `datetime` 对象
- `updateddate` 的值应该是一个 Python `datetime` 对象
- `enclosure` 的值应该是一个 `django.utils.feedgenerator.Enclosure` 实例
- `categories` 的值应该是一个 Unicode 对象序列

`SyndicationFeed.write()`

使用指定的编码把订阅源输出到类似文件的对象中。

`SyndicationFeed.writeString()`

以字符串的形式返回指定编码的订阅源。例如，下述代码创建一个 Atom 1.0 订阅源，然后打印到标准输出：

```
>>> from django.utils import feedgenerator
>>> from datetime import datetime
>>> f = feedgenerator.Atom1Feed(
...     title="My Weblog",
...     link="http://www.example.com/",
...     description="In which I write about what I ate today.",
...     language="en",
...     author_name="Myself",
...     feed_url="http://example.com/atom.xml")
>>> f.add_item(title="Hot dog today",
...     link="http://www.example.com/entries/1/",
...     pubdate=datetime.now(),
...     description="<p>Today I had a Vienna Beef hot dog. It was pink, plump and per\
fect.</p>")
>>> print(f.writeString('UTF-8'))
<?xml version="1.0" encoding="UTF-8"?>
<feed xmlns="http://www.w3.org/2005/Atom" xml:lang="en">
...
</feed>
```

自定义订阅源生成器

如果想生成自定义的订阅源格式，有几个选择。如果是全新的订阅源格式，可以编写一个 `SyndicationFeed` 的子类，把 `write()` 和 `writeString()` 方法完全替换掉。然而，如果订阅源格式派生自 RSS 或 Atom（如 GeoRSS、Apple 的 iTunes 播客格式，等等），有更好的选择。

这些订阅源类型往往为底层格式添加额外的元素和（或）属性，而 `SyndicationFeed` 为此提供了一系列方法。因此，可以创建适当订阅源生成器类（`Atom1Feed` 或 `Rss201rev2Feed`）的子类，然后扩展。这些方法是：

SyndicationFeed.root_attributes(self,)

返回一个属性字典，添加到订阅源的根元素（feed 或 channel）中。

SyndicationFeed.add_root_elements(self, handler)

向订阅源根元素（feed 或 channel）中添加元素的回调。handler 是一个 XMLGenerator（来自 Python 内置的 SAX 库），添加元素就是在它上面调用方法。

SyndicationFeed.item_attributes(self, item)

返回一个属性字典，添加到各个条目元素（item 或 entry）中。参数 item 是一个字典，包含传给 SyndicationFeed.add_item() 的所有数据。

SyndicationFeed.add_item_elements(self, handler, item)

把元素添加到各个条目（item 或 entry）中的回调。handler 和 item 同上。

> **提醒**
>
> 如果覆盖了这些方法，记得要调用超类中的方法，因为它们负责为各种订阅源格式添加所需的元素。

例如，可以像这样实现 iTunes RSS 订阅源生成器：

```
class iTunesFeed(Rss201rev2Feed):
    def root_attributes(self):
        attrs = super(iTunesFeed, self).root_attributes()
        attrs['xmlns:itunes'] =
          'http://www.itunes.com/dtds/podcast-1.0.dtd'
        return attrs

    def add_root_elements(self, handler):
        super(iTunesFeed, self).add_root_elements(handler)
        handler.addQuickElement('itunes:explicit', 'clean')
```

显然，完整的自定义订阅源类还有很多工作要做，但是上例应该能提供基本的思路。

14.7 网站地图框架

网站地图是网站中的一个 XML 文件，其作用是告诉搜索引擎索引程序网站中页面的更新频率和重要程度。这些信息有助于搜索引擎索引你的网站。网站地图的更多信息参见 sitemaps.org。

Django 的网站地图框架通过 Python 代码表述这些信息，自动生成网站地图 XML 文件。用起来跟 Django 的订阅源框架很像。若想创建网站地图，只需编写一个 Sitemap 子类，然后在 URL 配置中指向它。

14.7.1 安装

安装网站地图应用的步骤如下：

1. 把 "django.contrib.sitemaps" 添加到 INSTALLED_APPS 设置中。

2. 确保 TEMPLATES 设置中有 DjangoTemplates 后端，而且它的 APP_DIRS 选项设为 True。默认就有这个后端，除非修改了这个设置，否则不用动。
3. 安装网站框架。

14.7.2 初始化

若想在 Django 网站中生成网站地图，在 URL 配置中添加下述模式：

```
from django.contrib.sitemaps.views import sitemap

url(r'^sitemap\.xml$', sitemap, {'sitemaps': sitemaps},
    name='django.contrib.sitemaps.views.sitemap')
```

这个 URL 模式的作用是，当客户端访问 /sitemap.xml 时，让 Django 构建一个网站地图。网站地图文件的名称无关紧要，但是位置很重要。搜索引擎只索引网站地图中列出的当前及以下层级的 URL。例如，把 sitemap.xml 放在根目录中，可以引用网站中的任何 URL；而把网站地图放在 /content/sitemap.xml，则只能引用以 /content/ 开头的 URL。

网站地图视图有个额外的必须参数：{'sitemaps': sitemaps}。sitemaps 应该是个字典，把栏目标签（如 blog 或 news）映射到对应的 Sitemap 子类（如 BlogSitemap 或 NewsSitemap）上。此外，也可以映射到 Sitemap 子类的实例上（如 BlogSitemap(some_var)）。

14.7.3 网站地图类

网站地图类是表示网站地图中一个栏目下的条目的 Python 类。例如，可以让一个网站地图类表示博客中的所有文章，让另一个网站地图类表示日程表中的所有事项。

最简单的情况是把所有栏目放在同一个 sitemap.xml 文件中，不过也可以使用这个框架生成引用其他网站地图文件（一个栏目一个）的网站地图索引文件（参见 14.7.8 节）。

网站地图类必须是 django.contrib.sitemaps.Sitemap 的子类。可以放在代码基的任何位置。

14.7.4 一个简单的示例

假设有个博客系统，里面有个 Entry 模型，我们想在网站地图中列出指向各篇文章的链接。此时，可以这样定义网站地图类：

```
from django.contrib.sitemaps import Sitemap
from blog.models import Entry

class BlogSitemap(Sitemap):
    changefreq = "never"
    priority = 0.5

    def items(self):
        return Entry.objects.filter(is_draft=False)

    def lastmod(self, obj):
        return obj.pub_date
```

注意：

- 类属性 changefreq 和 priority 分别对应于 <changefreq> 和 <priority> 元素。如这里的 lastmod 一样，它们也可以定义为方法。
- items() 返回对象列表。返回的对象将传给网站地图属性对应的可调用方法（location、lastmod、changefreq 和 priority）。
- lastmod 方法应该返回一个 Python datetime 对象。
- 这里没有定义 location 方法，但是，如果想指定对象的 URL，可以定义。默认情况下，location() 返回在各个对象上调用 get_absolute_url() 得到的结果。

14.7.5 网站地图类 API

网站地图类可以定义下述方法和属性。

items

必须的。返回对象列表。网站地图框架不关心对象的类型，只负责把这些对象传给 location()、lastmod()、changefreq() 和 priority() 方法。

location

可选的。方法或属性。如果是方法，应该返回 items() 得到的对象的绝对路径；如果是属性，其值应该是一个字符串，表示 items() 返回的各个对象的绝对路径。

两种情况下，绝对路径指的都是不含协议和域名的 URL。例如：

- 正确：'/foo/bar/'
- 错误：'example.com/foo/bar/'
- 错误：'http://example.com/foo/bar/'

如果未提供 location，网站地图框架在 items() 返回的各个对象上调用 get_absolute_url() 方法。如果协议不是 http，使用 protocol 指定。

lastmod

可选的。方法或属性。如果是方法，应该接受一个参数，即 items() 返回的对象之一，返回那个对象的最后修改日期（时间，一个 Python datetime.datetime 对象）。

如果是属性，其值应该是一个 datetime.datetime 对象，表示 items() 返回的各个对象的最后修改日期（时间）。如果网站地图中的所有条目都有最后修改日期，views.sitemap() 生成的网站地图将把 Last-Modified 首部设为最近的最后修改日期。

可以激活 ConditionalGetMiddleware，让 Django 根据 If-Modified-Since 首部响应请求，在网站地图不变时不发送。

changefreq

可选的。方法或属性。如果是方法，应该接受一个参数，即 items() 返回的对象之一，返回一个 Python 字符串，表示那个对象的更新频率；如果是属性，其值应该是一个字符串，表示 items() 返回的各个对象的更新频率。

不管使用方法还是属性，changefreq 可以使用的值有：

- 'always'
- 'hourly'
- 'daily'
- 'weekly'
- 'monthly'
- 'yearly'
- 'never'

`priority`

可选的。方法或属性。如果是方法，应该接受一个参数，即 `items()` 返回的对象之一，返回一个字符串或浮点数，表示那个对象的优先级。

如果是属性，其值应该是一个字符串或浮点数，表示 `items()` 返回的每个对象的优先级。`priority` 的取值示例：`0.4`、`1.0`。页面的默认优先级是 `0.5`。详情参见 sitemaps.org 中的文档。

`protocol`

可选的。这个属性定义网站地图中的 URL 使用什么协议（`http` 或 `https`）。如果未设定，使用请求网站地图所用的协议。如果网站地图在请求上下文之外构建，默认使用 `http`。

`i18n`

可选的。这个布尔值属性定义是否使用 `LANGUAGES` 设置中的所有语言生成网站地图中的 URL。默认为 `False`。

14.7.6 快捷方式

网站地图框架为常见情况提供了一个便利的类——`django.contrib.syndication.GenericSitemap`。

使用这个类创建网站地图时，要传递一个字典，其中至少有一个 `queryset` 键，用于生成网站地图中的条目。此外，还可以提供 `date_field` 键，指定查询集合中各对象的日期使用哪个字段。日期字段中的值用于生成网站地图的 `lastmod` 属性。

还可以把 `priority` 和 `changefreq` 关键字参数传给 `GenericSitemap` 构造方法，指定各 URL 的对应属性。

示例

下述示例在 URL 配置中使用 `GenericSitemap`：

```
from django.conf.urls import url
from django.contrib.sitemaps import GenericSitemap
from django.contrib.sitemaps.views import sitemap
from blog.models import Entry

info_dict = {
    'queryset': Entry.objects.all(),
    'date_field': 'pub_date',
}
```

```
urlpatterns = [
    # 使用 info_dict 的通用视图
    # ...

    # 网站地图
    url(r'^sitemap\.xml$', sitemap,
        {'sitemaps': {'blog': GenericSitemap(info_dict, priority=0.6)}},
        name='django.contrib.sitemaps.views.sitemap'),
]
```

14.7.7 静态视图的网站地图

通常，我们还想让搜索引擎爬虫索引对象详情页面和普通页面之外的视图。为此，要在 `items` 中列出视图的 URL 名称，然后在网站地图的 `location` 方法中调用 `reverse()`。例如：

```
# sitemaps.py
from django.contrib import sitemaps
from django.core.urlresolvers import reverse

class StaticViewSitemap(sitemaps.Sitemap):
    priority = 0.5
    changefreq = 'daily'

    def items(self):
        return ['main', 'about', 'license']

    def location(self, item):
        return reverse(item)

# urls.py
from django.conf.urls import url
from django.contrib.sitemaps.views import sitemap

from .sitemaps import StaticViewSitemap
from . import views

sitemaps = {
    'static': StaticViewSitemap,
}

urlpatterns = [
    url(r'^$', views.main, name='main'),
    url(r'^about/$', views.about, name='about'),
    url(r'^license/$', views.license, name='license'),
    # ...
    url(r'^sitemap\.xml$', sitemap, {'sitemaps': sitemaps},
        name='django.contrib.sitemaps.views.sitemap')
]
```

14.7.8 创建网站地图索引

网站地图框架还支持创建网站地图索引，引用 `sitemaps` 字典中针对各个栏目的网站地图文件。这种方式的不

同之处是：

- 要在 URL 配置中使用两个视图：django.contrib.sitemaps.views.index() 和 django.contrib.sitemaps.views.sitemap()。
- django.contrib.sitemaps.views.sitemap() 视图接受一个 section 关键字参数。

对上述示例来说，相应的 URL 配置如下：

```
from django.contrib.sitemaps import views

urlpatterns = [
    url(r'^sitemap\.xml$', views.index, {'sitemaps': sitemaps}),
    url(r'^sitemap-(?P<section>.+)\.xml$', views.sitemap,
        {'sitemaps': sitemaps}),
]
```

这样配置之后会自动生成一个 sitemap.xml 文件，引用 sitemap-flatpages.xml 和 sitemap-blog.xml。网站地图类和 sitemaps 字典无需修改。

如果网站地图中的 URL 超过 50,000 个，应该创建索引文件。这样，Django 能自动为网站地图分页，而且在索引中能体现出来。如果不是直接使用网站地图视图，例如包装在缓存装饰器中，必须为网站地图视图命名，并把 sitemap_url_name 传给索引视图：

```
from django.contrib.sitemaps import views as sitemaps_views
from django.views.decorators.cache import cache_page

urlpatterns = [
    url(r'^sitemap\.xml$',
        cache_page(86400)(sitemaps_views.index),
        {'sitemaps': sitemaps, 'sitemap_url_name': 'sitemaps'}),
    url(r'^sitemap-(?P<section>.+)\.xml$',
        cache_page(86400)(sitemaps_views.sitemap),
        {'sitemaps': sitemaps}, name='sitemaps'),
]
```

14.7.9 自定义模板

如果想使用其他的模板生成网站地图或索引，在 URL 配置中把 template_name 参数传给 sitemap 和 index 视图：

```
from django.contrib.sitemaps import views

urlpatterns = [
    url(r'^custom-sitemap\.xml$', views.index, {
        'sitemaps': sitemaps,
        'template_name': 'custom_sitemap.html'
    }),
    url(r'^custom-sitemap-(?P<section>.+)\.xml$', views.sitemap, {
        'sitemaps': sitemaps,
        'template_name': 'custom_sitemap.html'
    }),
]
```

14.7.10 上下文变量

自定义 index() 和 sitemap() 视图的模板时，可以使用下述上下文变量。

索引

sitemaps 变量的值是一个列表，包含各个网站地图的绝对 URL。

网站地图

urlset 变量的值是一个列表，包含应该出现在网站地图中的 URL。各个 URL 都具有 Sitemap 类定义的属性：

- changefreq
- item
- lastmod
- location
- priority

各个 URL 多了个 item 属性，这么做为了更加方便定制模板，例如针对 Google 新闻的网站地图。假设网站地图类的 items() 方法返回的条目具有 publication_data 和 tags 字段，那么可以像下面这样生成兼容 Google 的网站地图：

```
{% spaceless %}
{% for url in urlset %}
    {{ url.location }}
    {% if url.lastmod %}{{ url.lastmod|date:"Y-m-d" }}{% endif %}
    {% if url.changefreq %}{{ url.changefreq }}{% endif %}
    {% if url.priority %}{{ url.priority }}{% endif %}

    {% if url.item.publication_date %}
        {{ url.item.publication_date|date:"Y-m-d" }}
    {% endif %}

    {% if url.item.tags %}{{ url.item.tags }}{% endif %}

{% endfor %}
{% endspaceless %}
```

14.7.11 通知 Google

网站地图有变化时，你可能想通知 Google，让它重新索引你的网站。为此，网站地图框架提供了一个函数。

django.contrib.syndication.ping_google()

ping_google() 函数有个可选的参数，其值为网站地图的绝对路径（如 '/sitemap.xml'）。如果未提供这个参数，ping_google() 尝试反向解析 URL 配置，找出网站地图的路径。如果无法确定网站地图的 URL，ping_google() 抛出 django.contrib.sitemaps.SitemapNotFound 异常。

可以在模型的 save() 方法中调用 ping_google() 函数：

```python
from django.contrib.sitemaps import ping_google

class Entry(models.Model):
    # ...
    def save(self, force_insert=False, force_update=False):
        super(Entry, self).save(force_insert, force_update)
        try:
            ping_google()
        except Exception:
            # 简化处理，因为可能出现多种与 HTTP 有关的异常
            pass
```

然而，更加有效的方式是在 cron 脚本或定时任务中调用 `ping_google()` 函数。这个函数会向 Google 的服务器发送 HTTP 请求，因此你可能不想每次执行 `save()` 的过程中都有网络开销。

使用 `manage.py` 通知 Google

在项目中添加网站地图应用之后，还可以使用管理命令 `ping_google` 通知 Google：

```
python manage.py ping_google [/sitemap.xml]
```

> **先在 Google 中注册**
>
> 在 Google Webmaster Tools 中注册你的网站之后才能使用 `ping_google` 命令。

14.8 接下来

我们将继续探索 Django 内置的工具，下一章深入讨论 Django 的会话框架。

第 15 章 Django 会话

试想一下，如果每访问网站中的一个页面都要重新登录，或者你经常访问的网站忘记了你的设置，每次访问都要重新输入，那该有多麻烦。

现今，如果网站不通过某种方式记住你是谁，以及你之前在网站中的活动情况，失去的是可用性和便利性。HTTP 本身是无状态的，两次请求之间没有连续性，服务器无法知晓后续请求是不是来自同一个人。

这种状态上的缺失由*会话*（session）弥补。会话是位于浏览器和 Web 服务器之间的半永久性双向通信。大多数情况下，访问现代的网站时，Web 服务器会使用*匿名会话*记录与访问有关的数据。这种会话之所以是匿名的，原因在于 Web 服务器只能记录你做了什么，却无法得知你是谁。

我们遇到过这种情况：一段时间之后再访问电商网站，你会发现，虽然没有提供个人信息，但是以前放在购物车中的商品还在。会话经常使用名声不好且很少被人理解的 cookie 持久存储。与其他所有 Web 框架一样，Django 也使用 cookie，但是你会发现，它的处理方式更灵巧、更安全。

Django 完全支持匿名会话。通过会话框架可以针对网站的每个访客存储和检索任意的数据。Django 把会话数据存储在服务器端，而且对发送和接收 cookie 的过程做了抽象。cookie 中存储的是会话 ID，而不是数据本身（除非使用基于 cookie 的后端）——这样实现更安全。

15.1 启用会话

会话由一个中间件实现。`django-admin startproject` 命令创建的 `settings.py` 文件默认已激活 `SessionMiddleware`。为了保证会话可用，`MIDDLEWARE_CLASSES` 设置中要列出 `'django.contrib.sessions.middleware.SessionMiddleware'`。

如果不想使用会话，可以把 `MIDDLEWARE_CLASSES` 中的 `SessionMiddleware` 和 `INSTALLED_APPS` 中的 `'django.contrib.sessions'` 删掉。这样能节省少量的开销。

15.2 配置会话引擎

默认情况下，Django 在数据库中存储会话（使用 `django.contrib.sessions.models.Session` 模型）。这样虽然方便，但是某些情况下在别处存储数据速度更快，因此 Django 允许修改配置，把会话数据存储在文件系统或缓存中。

15.2.1 使用基于数据库的会话

如果想把会话存储在数据库中，要在 `INSTALLED_APPS` 设置中添加 `'django.contrib.sessions'`。然后，运行 `manage.py migrate` 命令，创建存储会话数据的数据表。

15.2.2 使用基于缓存的会话

为了提升性能，你可能想使用基于缓存的会话后端。使用 Django 的缓存系统存储会话数据之前，要先配置好缓存，详情参见第 16 章。

> **提醒**
>
> 仅当使用 Memcached 这个缓存后端时才应该使用基于缓存的会话。本地内存缓存后端留存数据的时间不够长，不是个好选择；此外，直接使用文件或数据库存储会话，比一切都通过文件或数据库缓存后端发送更快。而且，本地内存缓存后端对多进程不安全，因此可能不适合在生产环境中使用。

如果 `CACHES` 设置定义了多个缓存后端，Django 将使用默认的那个。若想使用其他缓存后端，把 `SESSION_CACHE_ALIAS` 设为那个后端的名称。配置好缓存之后，在缓存中存储会话数据有两种方式：

1. 把 `SESSION_ENGINE` 设为 `"django.contrib.sessions.backends.cache"`，使用简单的缓存会话存储器。此时，会话数据直接存储在缓存中。然而，会话数据可能无法持久存储，因为缓存存储器空间用完或缓存服务器重启后数据会丢失。
2. 若想持久存储缓存的会话数据，把 `SESSION_ENGINE` 设为 `"django.contrib.sessions.backends.cached_db"`。此时使用的是*直写式缓存*（write-through cache），写入缓存的数据也会写入数据库。如果缓存中没有会话数据，再到数据库中读取。

这两种会话存储器的速度都十分快，不过简单缓存更快，因为它不做持久存储。多数情况下，`cached_db` 后端足够快了；然而，如果你还需要那最后一点性能，而且想随着时间的推移擦除会话数据，可以使用 `cache` 后端。如果使用 `cached_db` 后端，还要按照 15.2.1 节的说明配置。

15.2.3 使用基于文件的会话

若想使用基于文件的会话，把 `SESSION_ENGINE` 设为 `"django.contrib.sessions.backends.file"`。你可能还想配置 `SESSION_FILE_PATH`（默认为 `tempfile.gettempdir()` 得到的结果，通常是 `/tmp`），设定 Django 存储会话文件的位置。确保 Web 服务器有指定位置的读写权限。

15.2.4 使用基于 cookie 的会话

若想使用基于 cookie 的会话，把 `SESSION_ENGINE` 设为 `"django.contrib.sessions.backends.signed_cookies"`。会话数据使用 Django 的签名工具和 `SECRET_KEY` 存储。

建议别动 `SESSION_COOKIE_HTTPONLY` 设置，仍然使用 `True`，以防 JavaScript 访问存储的数据。

> **提醒**
>
> 如果没有保护好 `SECRET_KEY`，而且使用的是 `PickleSerializer`，那么可能导致远程执行任意代码。
>
> 获得 `SECRET_KEY` 的攻击者不仅可以篡改会话数据（能通过网站的验证），而且可以远程执行任意代码，因为数据是使用 pickle 序列化的。使用基于 cookie 的会话时一定要多加小心，保护好密钥，不让任何系统远程访问。

会话数据有签名，但是未加密

cookie 后端中的会话数据可以被客户端读取。为了防止客户端修改数据，cookie 使用 MAC（Message Authentication Code，消息验证码）保护，因此，一旦被篡改，会话数据即行失效。如果 cookie 无法存储全部会话数据，有数据丢失，会话也失效。虽然 Django 会压缩数据，但是也完全有可能超出 cookie 的 4096 字节限制。

无法保证新鲜

还要注意，虽然 MAC 可以保证数据的真伪（即由你的网站生成，而不是别人生成的）、数据的完整性（即数据都在，而且正确），但是无法保障数据是新鲜的，即发回的数据与发送给客户端的一样。这意味着，某些情况下 cookie 后端中的会话数据可能导致*重放攻击*（replay attack）。有些会话后端会在服务器端记录各个会话，并在用户退出后让会话失效，但是基于 cookie 的会话则不然。因此，攻击者盗取用户的 cookie 后，即使用户已经退出，也能以那个用户的身份登录。仅当 cookie 的存在时间比 `SESSION_COOKIE_AGE` 设定的值长时才会被识别为“过期的”。

最后要说的是，如果上述警告还没有让你放弃使用基于 cookie 的会话，那么再看这一点：cookie 的大小可能会对网站的速度有影响。

15.3 在视图中使用会话

激活 `SessionMiddleware` 之后，每个 `HttpRequest` 对象（传给任何 Django 视图函数的第一个参数）都有一个 `session` 属性，其值是一个类似字典的对象。在视图中，任何时候都可以读写 `request.session`，甚至可以多次编辑。

会话对象继承自基类 `backends.base.SessionBase`，具有下述标准的字典方法：

- `__getitem__(key)`
- `__setitem__(key, value)`
- `__delitem__(key)`
- `__contains__(key)`
- `get(key, default=None)`
- `pop(key)`
- `keys()`
- `items()`
- `setdefault()`
- `clear()`

此外，还有下述方法。

15.3.1 flush()

从会话中删除当前会话数据，并把会话 cookie 删除。如果想确保用户的浏览器不能再访问之前的会话数据（例如 `django.contrib.auth.logout()` 函数会调用它），可以调用这个方法。

15.3.2 set_test_cookie()

设定一个测试 cookie，确认用户的浏览器是否支持 cookie。由于 cookie 运作方式上的限制，只能等到用户下一次请求页面时才能做这项测试。详情参见 15.6 节。

15.3.3 test_cookie_worked()

根据用户的浏览器是否接受测试 cookie 返回 True 或 False。由于 cookie 运作方式上的限制，要在之前的单独页面请求中调用 set_test_cookie()。详情参见 15.6 节。

15.3.4 delete_test_cookie()

删除测试 cookie。测试完用于清理。

15.3.5 set_expiry(value)

设定会话的过期时间。可以传入的值有：

- 把 value 设为整数时，会话在指定的秒数后过期。例如，request.session.set_expiry(300) 的意思是会话在 5 分钟后过期。
- 把 value 设为 datetime 或 timedelta 对象时，会话在指定的日期（时间）过期。注意，仅当使用 PickleSerializer 时 datetime 和 timedelta 对象才可以序列化。
- 把 value 设为 0 时，用户的会话 cookie 在 Web 浏览器关闭后过期。
- 把 value 设为 None 时，使用全局的会话过期策略。

读取会话不会延长过期时间。会话过期的依据是最后的修改时间。

15.3.6 get_expiry_age()

返回会话还有多少秒过期。对未自定义过期时间（或者在浏览器关闭时过期）的会话来说，返回的值等于 SESSION_COOKIE_AGE。这个方法接受两个可选的关键字参数：

- modification：一个 datetime 对象，会话的最后修改时间。默认为当前时间。
- expiry：表示会话过期信息的 datetime 对象、整数（秒数）或 None。如果调用 set_expiry() 设定了过期信息，默认就是那个值，否则是 None。

15.3.7 get_expiry_date()

返回会话过期的日期。对未自定义过期时间（或者在浏览器关闭时过期）的会话来说，返回 SESSION_COOKIE_AGE 的值距现在的秒数。这个方法也接受 get_expiry_age() 方法的那两个关键字参数。

15.3.8 get_expire_at_browser_close()

根据用户的会话 cookie 是否在 Web 浏览器关闭后过期，返回 True 或 False。

15.3.9 clear_expired()

从会话存储器中移除过期的会话。clearsessions 命令会调用这个类方法。

15.3.10 cycle_key()

创建一个新的会话键，但是保留当前的会话数据。`django.contrib.auth.login()` 会调用这个方法，以防*会话固定*（session fixation）攻击。

15.4 会话对象指导方针

- `request.session` 字典的键使用普通的 Python 字符串。这不是不得违反的规则，而是一种约定。
- 以下划线开头的键是保留的，供 Django 内部使用。
- 别使用新对象覆盖 `request.session`，也别访问或设定它的属性。像 Python 字典那样使用它。

15.5 会话序列化

在 1.6 版之前，Django 默认使用 `pickle` 序列化会话数据，然后再存入后端。如果使用的是签名的 cookie 会话后端，而且攻击者知道了 `SECRET_KEY`（Django 没有内在的漏洞会导致这个密钥泄露），那么攻击者可以在会话中插入字符串，反序列化会话后在服务器中执行任意的代码。网上充斥着这种简单而易行的技术。

虽然为了防止篡改，cookie 中存储的是签名后的会话数据，可是一旦 `SECRET_KEY` 泄露，远程执行代码的漏洞就凸显出来了。把序列化会话数据的方式由 `pickle` 改成 JSON 可以降低这种攻击的可行性。为此，Django 1.5.3 引入了一个新设置，即 `SESSION_SERIALIZER`，用于自定义会话的序列化格式。为了向后兼容，Django 1.5.x 默认使用 `django.contrib.sessions.serializers.PickleSerializer`；但是，为了增强安全性，Django 1.6 之后默认使用 `django.contrib.sessions.serializers.JSONSerializer`。

虽然 15.5.2 节将指出 JSON 序列化的一些不足，但是我们还是强烈推荐始终使用 JSON 序列化，尤其是使用 cookie 后端时。

15.5.1 内置的序列化程序

`serializers.JSONSerializer`

`django.core.signing` 提供的 JSON 序列化程序包装。只能序列化基本的数据类型。此外，JSON 只支持字符串键，因此不能在 `request.session` 中使用非字符串键：

```
>>> # 初始赋值
>>> request.session[0] = 'bar'
>>> # 后续请求要序列化和反序列化会话数据
>>> request.session[0]  # KeyError
>>> request.session['0']
'bar'
```

JSON 序列化的不足之处参见 15.5.2 节。

`serializers.PickleSerializer`

支持序列化任何 Python 对象，但是如前所述，如果攻击者得知了 `SECRET_KEY`，会暴露远程代码执行漏洞。

15.5.2 自己编写序列化程序

注意，与 `PickleSerializer` 不同，`JSONSerializer` 无法处理所有 Python 数据类型。通常，在便利和安全之间，我们需要取舍。如果想在 JSON 序列化的会话中存储复杂的数据类型，例如 `datetime` 和 `Decimal`，需要

自己编写序列化程序（或者在存入 request.session 之前把复杂的值转换成 JSON 能序列化的对象）。

虽然序列化复杂的值很简单（django.core.serializers.json.DateTimeAwareJSONEncoder 可以助你一臂之力），但是解码时若想得到与序列化之前完全一样的值并非易事。例如，可能会把碰巧与日期格式一样的字符串变成 datetime 对象。

自定义的序列化程序类必须实现两个方法：dumps(self, obj) 和 loads(self, data)。它们分别用于序列化和反序列化会话数据字典。

15.6 设定测试 cookie

为了便于测试用户的浏览器是否支持 cookie，Django 提供了一种简单的方式：只需在一个视图中调用 request.session 的 set_test_cookie() 方法，然后在后续视图（与前一个不同）中调用 test_cookie_worked() 方法。

把 set_test_cookie() 和 test_cookie_worked() 分开看似不便，但却是必要的，因为 cookie 就是这样运作的。设定 cookie 后无法立即判断浏览器是否能接受，要等到下一次请求才能确定。测试完之后，最好调用 delete_test_cookie() 方法清理一下。

下面是常见的做法举例：

```
def login(request):
    if request.method == 'POST':
        if request.session.test_cookie_worked():
            request.session.delete_test_cookie()
            return HttpResponse("You're logged in.")
        else:
            return HttpResponse("Please enable cookies and try again.")
    request.session.set_test_cookie()
    return render_to_response('foo/login_form.html')
```

15.7 在视图之外使用会话

本节的示例直接从 django.contrib.sessions.backends.db 后端导入 SessionStore 对象。实际使用中，应该像下面这样从 SESSION_ENGINE 设定的会话引擎中导入 SessionStore：

```
>>> from importlib import import_module
>>> from django.conf import settings
>>> SessionStore = import_module(settings.SESSION_ENGINE).SessionStore
```

在视图之外处理会话数据也有 API：

```
>>> from django.contrib.sessions.backends.db import SessionStore
>>> s = SessionStore()
>>> # 存储距 Unix 纪元的秒数，因为 datetime 对象不能序列化成 JSON
>>> s['last_login'] = 1376587691
>>> s.save()
>>> s.session_key
'2b1189a188b44ad18c35e113ac6ceead'

>>> s = SessionStore(session_key='2b1189a188b44ad18c35e113ac6ceead')
>>> s['last_login']
```

```
1376587691
```

为了规避会话固定攻击，不存在的会话键会重新生成：

```
>>> from django.contrib.sessions.backends.db import SessionStore
>>> s = SessionStore(session_key='no-such-session-here')
>>> s.save()
>>> s.session_key
'ff882814010ccbc3c870523934fee5a2'
```

对 django.contrib.sessions.backends.db 后端来说，各个会话就是普通的 Django 模型。Session 模型在 django/contrib/sessions/models.py 中定义。鉴于此，我们可以使用常规的 Django 数据库 API 访问会话：

```
>>> from django.contrib.sessions.models import Session
>>> s = Session.objects.get(pk='2b1189a188b44ad18c35e113ac6ceead')
>>> s.expire_date
datetime.datetime(2005, 8, 20, 13, 35, 12)
```

注意，要调用 get_decoded() 获取会话字典。之所以要多这一步，是因为会话字典是以编码后的格式存储的：

```
>>> s.session_data
'KGRwMQpTJ19hdXRoX3VzZXJfaWQnCnAyCkkxCnMuMTExY2ZjODI2Yj...'
>>> s.get_decoded()
{'user_id': 42}
```

15.8 何时保存会话

默认情况下，仅当修改会话后才会将其存入会话数据库。这里说的"修改"是指会话字典中的值有增删：

```
# 修改了会话
request.session['foo'] = 'bar'

# 修改了会话
del request.session['foo']

# 修改了会话
request.session['foo'] = {}

# 小心：没有修改会话，因为修改的是 request.session['foo'] 而不是 request.session
request.session['foo']['bar'] = 'baz'
```

对上例的最后一步来说，可以设定会话对象的 modified 属性，明确表明修改了会话：

```
request.session.modified = True
```

若想修改这样的默认行为，把 SESSION_SAVE_EVERY_REQUEST 设为 True。此时，每次请求时 Django 都会把会话存入数据库。注意，只有创建或修改了会话才会发送会话 cookie。如果把 SESSION_SAVE_EVERY_REQUEST 设为 True，每次请求都会发送会话 cookie。类似地，每次发送会话 cookie 都会更新 expires 部分。如果响应的状态码是 500，不会保存会话。

15.9 持续到浏览器关闭的会话与持久会话

可以使用 SESSION_EXPIRE_AT_BROWSER_CLOSE 设置控制会话框架使用持续到浏览器关闭的会话还是持久会话。SESSION_EXPIRE_AT_BROWSER_CLOSE 的默认值是 False，这意味着会话 cookie 在用户的浏览器中存储的时间等于 SESSION_COOKIE_AGE。如果不想让用户每次打开浏览器后都重新登录，使用这个默认值。

如果把 SESSION_EXPIRE_AT_BROWSER_CLOSE 设为 True，Django 将使用持续到浏览器关闭的会话，即用户关闭浏览器后会话 cookie 即告过期。

> **提示**
>
> 有些浏览器（例如 Chrome）提供了这样的设置，允许用户重新打开浏览器后继续使用会话。某些情况下，这种行为与 SESSION_EXPIRE_AT_BROWSER_CLOSE 设置相抵，浏览器关闭后会话不过期。测试启用了 SESSION_EXPIRE_AT_BROWSER_CLOSE 的 Django 应用时要注意。

15.10 清理会话存储器

用户创建的会话不断积累，占据着会话存储器。Django 没有提供自动清除过期会话的机制，因此你自己要定期清除过期会话。为此，Django 提供了一个清理命令：clearsessions。建议你定期执行这个命令，比如说定义一个每天执行的 cron 作业。

注意，缓存后端不受这个问题的影响，因为缓存会自动删除过期的数据。cookie 亦然，因为会话数据由用户的浏览器存储。

15.11 接下来

接下来继续探讨高级的 Django 话题，这一次要探讨的是 Django 的缓存后端。

第 16 章 Django 的缓存框架

动态网站的不足之处体现在“动态”上。每请求一个页面，Web 服务器都要做各种计算，为了让访客看到页面，要查询数据库、渲染模板，还要执行一些业务逻辑。从消耗方面来看，这个过程比从文件系统中读取一个文件要耗资源。

对多数 Web 应用程序来说，这个过程消耗的资源其实并不多。没有几个网站能有华盛顿邮报网站或 Slashdot 那样的体量，多数网站是中小型的，流量水平一般。但是对中高流量水平的网站来说，一定要从根本上尽量降低消耗。

缓存就是为之而生的。存入缓存是指把耗资源的计算结果保存起来，不用每次都重新计算。下面是一段伪代码，说明缓存在动态生成网页的过程中所起的作用：

```
given a URL, try finding that page in the cache if the page is in the cache:
    return the cached page
else:
    generate the page
    save the generated page in the cache (for next time)
    return the generated page
```

Django 提供了一个强健的缓存系统，能把动态页面保存起来，不用每次请求都重新生成。为了便于使用，Django 提供的缓存分为不同的粒度层次，可以缓存特定视图的输出、可以只缓存不易生成的片段，也可以缓存整个网站。

Django 还能很好地支持下游缓存，例如 Squid 和基于浏览器的缓存。这两种缓存不直接受你控制，但是你可以给出提示（通过 HTTP 首部），指明网站的哪些部分要缓存，以及如何缓存。

16.1 配置缓存

缓存系统要少许配置之后才能使用。具体而言，你要指明缓存的数据存储在哪里，在数据库中、文件系统中，还是直接放在内存中。这是一项重要决策，影响着缓存的性能。

缓存在设置文件中的 `CACHES` 设置项目中配置。

16.1.1 Memcached

Django 原生支持的速度最快的，也是效率最高的缓存是 Memcached。这是一个完全基于内存的缓存服务器，由 Danga Interactive 公司开发，最初的目的是处理 LiveJournal.com 的高负载，而后开源了。Facebook 和 Wikipedia 等网站使用它减少数据库访问，大幅提升网站的性能。

Memcached 以守护进程的方式运行，需要专门分配一定量的 RAM。它的作用很简单，就是为增加、读取和删除缓存中的数据提供高速接口。所有数据都直接存储在内存中，因此对数据库或文件系统没有消耗。

安装完 Memcached 之后，还要安装一个 Memcached 绑定。Python 的 Memcached 绑定有好几个，其中 python-memcached 和 python-memcached 是最常用的。在 Django 中使用 Memcached 的步骤如下：

- 把 `BACKEND` 设为 `django.core.cache.backends.memcached.MemcachedCache` 或 `django.core.cache.back-`

ends.memcached.PyLibMCCache（根据你所选的 Memcached 绑定而定）。

- 把 LOCATION 设为 ip:port 这种形式的值，其中 ip 是 Memcached 守护进程的 IP 地址，port 是运行 Memcached 的端口；或者设为 unix:path 这种形式的值，其中 path 是 Memcached 的 Unix 套接字文件的路径。

在下述示例中，Memcached 运行在本地（127.0.0.1）的 11211 端口上，使用的是 python-memcached 绑定：

```
CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.memcached.MemcachedCache',
        'LOCATION': '127.0.0.1:11211',
    }
}
```

在下述示例中，Memcached 通过一个本地的 Unix 套接字文件（/tmp/memcached.sock）访问，使用的是 python-memcached 绑定：

```
CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.memcached.MemcachedCache',
        'LOCATION': 'unix:/tmp/memcached.sock',
    }
}
```

Memcached 有个极好的特性：支持多个服务器共享同一个缓存。这意味着，可以在多台设备中运行多个 Memcached 守护进程，而程序把这一组设备视作一个缓存，因此无需在每台设备中重复缓存值。若想使用这个特性，在 LOCATION 中列出全部服务器的地址，使用分号分隔，或者以列表指定。

在下述示例中，缓存在 172.19.26.240 和 172.19.26.242 两个 IP 地址（端口都是 11211）对应的设备中运行的 Memcached 实例之间共享：

```
CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.memcached.MemcachedCache',
        'LOCATION': [
            '172.19.26.240:11211',
            '172.19.26.242:11211',
        ]
    }
}
```

在下述示例中，缓存在 172.19.26.240（11211 端口）、172.19.26.242（11212 端口）和 172.19.26.244（11213 端口）三个 IP 地址对应的设备中运行的 Memcached 实例之间共享：

```
CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.memcached.MemcachedCache',
        'LOCATION': [
            '172.19.26.240:11211',
            '172.19.26.242:11212',
            '172.19.26.244:11213',
        ]
    }
```

```
}
```

最后要指出的是，Memcached 这种基于内存的缓存有个缺点：因为缓存的数据存储在内存中，所以服务器一旦崩溃，数据就不复存在了。

显然，内存的目的不是永久存储数据，因此别把基于内存的缓存当做唯一的存储器。毋庸置疑，Django 支持的所有缓存后端都不应该用于永久存储，它们的唯一目的是缓存，而不是存储。之所以在这里特别指出，是因为基于内存的缓存更是临时的。

16.1.2 数据库缓存

Django 可以把缓存数据存入数据库。如果数据库服务器的索引稳定快速，这样做效果最好。以数据表为缓存后端的步骤如下：

- 把 `BACKEND` 设为 `django.core.cache.backends.db.DatabaseCache`。
- 把 `LOCATION` 设为数据表的名称。这个数据表的名称随意起，只要是有效的表名，而且没被数据库占用。

在下述示例中，缓存表的名称是 `my_cache_table`：

```
CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.db.DatabaseCache',
        'LOCATION': 'my_cache_table',
    }
}
```

创建缓存表

使用数据库缓存之前，必须执行下述命令创建缓存表：

```
python manage.py createcachetable
```

这个命令使用 Django 的数据库缓存系统所期待的格式在数据库中创建一个表。表名从 `LOCATION` 中获取。如果使用多个数据库缓存，`createcachetable` 命令会分别为各个缓存创建一个表。如果使用多个数据库，`createcachetable` 命令会观察数据库路由的 `allow_migrate()` 方法（参见下文）。与 `migrate` 一样，`createcachetable` 命令不会触碰现有的表，而只是创建缺失的表。

多个数据库

如果把缓存存入多个数据库，还要为缓存表提供路由指令。为此，缓存表对应的模型名为 `CacheEntry`，位于名为 `django_cache` 的应用中。这个模型不出现在模型缓存中，它的作用是提供详细的路由信息。

例如，下述路由把读缓存操作交给 `cache_replica` 数据库处理，把写缓存操作交给 `cache_primary` 数据库处理，而且缓存表只向 `cache_primary` 中同步：

```
class CacheRouter(object):
    """A router to control all database cache operations"""

    def db_for_read(self, model, **hints):
        # 缓存从这个副本中读取
        if model._meta.app_label in ('django_cache',):
            return 'cache_replica'
```

```
        return None

    def db_for_write(self, model, **hints):
        # 缓存写入这个主缓存
        if model._meta.app_label in ('django_cache',):
            return 'cache_primary'
        return None

    def allow_migrate(self, db, model):
        # 只在主缓存中安装缓存模型
        if model._meta.app_label in ('django_cache',):
            return db == 'cache_primary'
        return None
```

如果不为数据库缓存模型指定路由方向，缓存后端将使用 `default` 数据库。当然，如果你不使用数据库缓存后端，就无需费心为数据库缓存模型提供路由指令。

16.1.3 文件系统缓存

基于文件的后端会序列化各个缓存的值，将其存入单独的文件。若想使用这个后端，把 `BACKEND` 设为 `'django.core.cache.backends.filebased.FileBasedCache'`，并把 `LOCATION` 设为合适的目录。

假如想把缓存的数据存储在 `/var/tmp/django_cache` 目录中，使用下述设置：

```
CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.filebased.FileBasedCache',
        'LOCATION': '/var/tmp/django_cache',
    }
}
```

使用 Windows 时，要在路径的开头加上盘符，例如：

```
CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.filebased.FileBasedCache',
        'LOCATION': 'c:/foo/bar',
    }
}
```

目录的路径应该是绝对的，即应该从文件系统的根开始。路径末尾有没有斜线没有关系。确保 `LOCATION` 指向的目录存在，而且运行 Web 服务器的用户有读写权限。以上例为例，如果运行服务器的用户是 `apache`，那么要确保 `/var/tmp/django_cache` 目录存在，而且 `apache` 用户有读写权限。

16.1.4 本地内存缓存

如果没在设置文件中配置，默认使用这个缓存。如果想要内存缓存的速度优势，但是无法使用 Memcached，可以考虑使用本地内存缓存后端。若想使用它，把 `BACKEND` 设为 `django.core.cache.backends.locmem.LocMemCache`。例如：

```
CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.locmem.LocMemCache',
```

```
        'LOCATION': 'unique-snowflake'
    }
}
```

LOCATION 用于标识各个内存存储器。如果只有一个本地内存缓存，可以不设 LOCATION；然而，如果有多个，至少要为其中一个命名，以示区分。

注意，每个进程都有各自私有的缓存实例。这意味着，缓存不能跨进程。这也意味着，本地内存缓存对内存的利用效率并不高，因此可能不适合在生产环境使用。在开发环境中用着还是不错的。

16.1.5 虚拟缓存（供开发）

最后，Django 还提供了一个虚拟缓存，它并不真正缓存，只是实现了缓存接口。如果生产环境大量使用缓存，而在开发和测试环境中不想使用缓存，又不想改代码，就可以使用这个后端。启用虚拟缓存的方式是像这样设置 BACKEND：

```
CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.dummy.DummyCache',
    }
}
```

16.1.6 使用自定义的缓存后端

尽管 Django 原生支持多个缓存后端，但有时候你可能想使用自定义的后端。此时，要把 CACHES 设置的 BACKEND 选项设为外部缓存后端的 Python 导入路径：

```
CACHES = {
    'default': {
        'BACKEND': 'path.to.backend',
    }
}
```

自定义缓存后端时，可以参照标准的后端。相关的源码在 django/core/cache/backends/ 目录中。

> **提示**
>
> 除非迫不得已，例如主机不支持，否则应该始终使用 Django 自带的缓存后端。内置的后端有良好的测试，而且易于使用。

16.1.7 缓存的参数

每个缓存后端都可以指定额外的参数，用于控制缓存的行为。额外的参数通过 CACHES 设置中额外的键指定。可用的参数有：

- TIMEOUT：缓存默认的超时时间（秒数）。默认值为 300 秒（5分钟）。可以设为 None，即缓存键永不过期。设为 0 时，缓存键立即过期（相当于没缓存）。
- OPTIONS：传给缓存后端的选项。每个后端可用的选项有所不同。以第三方库为支持的缓存后端会把选项直接传给底层的缓存库。
- 实现剔除策略的缓存后端（即本地内存后端、文件系统后端和数据库后端）接受下述选项：

    - MAX_ENTRIES：删除旧值之前允许存储的条目最大数。默认值为 300。
    - CULL_FREQUENCY：达到 MAX_ENTRIES 设定的上限时剔除的条目比例。真正的比例其实是 1 / CULL_FREQUENCY，因此把 CULL_FREQUENCY 设为 2 时，达到上限后剔除一半的条目。这个参数的值应该是一个整数，默认为 3。设为 0 时，达到上限后清空整个缓存。在某些后端中（尤其是数据库后端），这样做的速度更快，但代价是有更多的缓存缺失。

- KEY_PREFIX：Django 服务器自动在所有缓存键中加入（默认放在开头）的一个字符串。
- VERSION：Django 服务器生成的缓存键的默认版本号。
- KEY_FUNCTION：表示一个函数的点分路径字符串。对应的函数定义如何由前缀、版本号和键构成最终的缓存键。

下述示例配置一个文件系统后端，把超时时间设为 60 秒，把最大容量设为 1000 条：

```
CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.filebased.FileBasedCache',
        'LOCATION': '/var/tmp/django_cache',
        'TIMEOUT': 60,
        'OPTIONS': {'MAX_ENTRIES': 1000}
    }
}
```

16.2 整站缓存

配置好缓存之后，使用缓存最简单的方式是缓存整个网站。为此，要把 'django.middleware.cache.UpdateCacheMiddleware' 和 'django.middleware.cache.FetchFromCacheMiddleware' 添加到 MIDDLEWARE_CLASSES 设置中，如下所示：

```
MIDDLEWARE_CLASSES = [
    'django.middleware.cache.UpdateCacheMiddleware',
    'django.middleware.common.CommonMiddleware',
    'django.middleware.cache.FetchFromCacheMiddleware',
]
```

提醒

注意，我没写错，UpdateCacheMiddleware 必须放在列表的开头，而且 FetchFromCacheMiddleware 必须放在列表的末尾。具体原因有点费解，如果你想全面了解，请阅读 17.5 节。

然后，把下述必须的设置添加到 Django 设置文件中：

- CACHE_MIDDLEWARE_ALIAS：存储时使用的缓存别名。
- CACHE_MIDDLEWARE_SECONDS：每个页面要缓存的秒数。
- CACHE_MIDDLEWARE_KEY_PREFIX：如果缓存在使用同一个 Django 的多个网站中共享，把这个设置设为网站的名称，或者设为对 Django 实例来说唯一的字符串，以防键有冲突。如果不介意，设为空字符串。

FetchFromCacheMiddleware 缓存请求和首部允许缓存的，而且状态码为 200 的 GET 和 HEAD 请求的响应。针对相同 URL 的请求，如果查询参数有差，响应视为不同的，分开缓存。 这个中间件期望 HEAD 请求的响应首部与相应的 GET 请求一样，这样遇到 HEAD 请求就能返回缓存的 GET 响应。此外，UpdateCacheMiddleware 会自动

为各个 HttpResponse 对象设定几个首部：

- 请求页面的全新（未缓存）版时，把 Last-Modified 首部设为当前日期和时间。
- 把 Expires 首部设为当前日期和时间减去 CACHE_MIDDLEWARE_SECONDS 后得到的值。
- 设定 Cache-Control 首部，为页面指定最长有效期。同样，也要减去 CACHE_MIDDLEWARE_SECONDS 设置。

如果视图设定了缓存的过期时间（即 Cache-Control 首部中有 max-age 部分），页面的缓存将持续到那个时间，而不考虑 CACHE_MIDDLEWARE_SECONDS。使用 django.views.decorators.cache 中的装饰器可以轻易设定视图的过期时间（使用 cache_control 装饰器），或者禁用缓存（使用 never_cache 装饰器）。这些装饰器的详细说明参见 16.8 节。

如果 USE_I18N 设置的值为 True，生成的缓存键包含当前语言的名称。这样无需人工干预就能缓存多语言网站。

USE_L10N 设置的值为 True 时缓存键中也包含当前语言的名称。USE_TZ 设置的值为 True 时，缓存键中还包含当前时区。

16.3 视图层缓存

缓存框架更细粒度的使用方式是缓存单个视图的输出。django.views.decorators.cache 定义的 cache_page 装饰器可以自动缓存视图的响应，使用方法很简单：

```
from django.views.decorators.cache import cache_page

@cache_page(60 * 15)
def my_view(request):
    ...
```

cache_page 只有一个参数，用于设定缓存超时的秒数。在上例中，my_view() 视图的结果缓存 15 分钟。（注意，为了便于理解，我用的是 60 * 15，即 15 个 60 秒，等于 900。）

视图层的缓存与整站缓存一样，一个 URL 对应一个键。如果多个 URL 指向同一个视图，每个 URL 单独缓存。仍以 my_view 视图为例，如果 URL 配置是这样的：

```
urlpatterns = [
    url(r'^foo/([0-9]{1,2})/$', my_view),
]
```

那么针对 /foo/1/ 和 /foo/23/ 的请求分开缓存。一旦请求了特定的 URL（如 /foo/23/），后续对那个 URL 的请求将使用缓存。

cache_page 还接受一个可选的关键字参数 cache，指定使用特定的缓存（CACHES 设置中的某一个）存储结果。

默认使用 default 缓存，不过你可以根据需要任意指定：

```
@cache_page(60 * 15, cache="special_cache")
def my_view(request):
    ...
```

此外，在视图层还可以覆盖缓存前缀。cache_page 的可选关键字参数 key_prefix 作用与中间件的 CACHE_MIDDLEWARE_KEY_PREFIX 设置一样。下面举个例子：

```
@cache_page(60 * 15, key_prefix="site1")
def my_view(request):
    ...
```

key_prefix 和 cache 参数可以同时指定。key_prefix 参数的值将和 CACHES 设置中指定的 KEY_PREFIX 拼接在一起。

16.3.1 在 URL 配置中指定视图层缓存

前面几个缓存视图的例子是硬编码的，因为 cache_page 将就地更改 my_view 函数。这么做把视图和缓存系统耦合在一起了，并不妥当。原因有几方面。例如，你可能想在另一个不用缓存的网站中复用视图函数，或者想把视图提供给不需要缓存的人使用。

这些问题的解决方法是在 URL 配置中指定视图层缓存，而不在视图函数上指定。方法很简单，在 URL 配置中使用 cache_page 包装视图函数。

下面是前述示例中的 URL 配置：

```
urlpatterns = [
    url(r'^foo/([0-9]{1,2})/$', my_view),
]
```

下面的 URL 配置与之等效，不过 my_view 包装在 cache_page 中：

```
from django.views.decorators.cache import cache_page

urlpatterns = [
    url(r'^foo/([0-9]{1,2})/$', cache_page(60 * 15)(my_view)),
]
```

16.4 模板片段缓存

如果想更进一步控制，还可以使用 cache 模板标签缓存模板片段。为了能在模板中使用这个标签，在靠近模板顶部的位置写上 {% load cache %}。{% cache %} 模板标签缓存块中的内容一段时间。

至少要为 {% cache %} 标签提供两个参数，一个设定缓存超时的秒数，另一个为缓存的片段命名。这个名称原封不动，不视作变量。

例如：

```
{% load cache %}
{% cache 500 sidebar %}
    .. sidebar ..
{% endcache %}
```

有时可能想根据动态片段中的内容缓存多份。例如，分别为网站中的各个用户缓存上例中的侧边栏。为此，再给 {% cache %} 模板标签传入一个参数，用于唯一标识缓存片段：

```
{% load cache %}
{% cache 500 sidebar request.user.username %}
    .. sidebar for logged in user ..
{% endcache %}
```

为了标识片段，完全可以传入多个参数。直接把额外的参数传给 {% cache %} 标签即可。如果把 USE_I18N 设

为 True，整站缓存中间件将考虑当前语言。

在 cache 模板标签中可以使用一个翻译专用的变量达到同样的效果：

```
{% load i18n %}
{% load cache %}

{% get_current_language as LANGUAGE_CODE %}

{% cache 600 welcome LANGUAGE_CODE %}
    {% trans "Welcome to example.com" %}
{% endcache %}
```

缓存超时时间可以通过模板变量指定，只要那个变量最终得到的值是一个整数就行。

例如，如果 my_timeout 模板变量的值是 600，那么下面两行代码是等效的：

```
{% cache 600 sidebar %} ... {% endcache %}
{% cache my_timeout sidebar %} ... {% endcache %}
```

这样可以避免模板中有重复。你可以在一处定义一个变量，指定超时时间，然后在多个地方复用。cache 标签默认尝试使用名为 template_fragments 的缓存。如果这个缓存不存在，使用 default 缓存。可以使用 using 关键字参数（必须是最后一个参数）选择缓存后端。

```
{% cache 300 local-thing ...  using="localcache" %}
```

指定未配置的缓存会出错。

如果想获取缓存片段所用的缓存键，使用 make_template_fragment_key(fragment_name, vary_on=None) 函数。fragment_name 的值等于 cache 模板标签的第二个参数；vary_on 是一个列表，包含传给 cache 标签的所有额外参数。可以使用这个函数撤销或覆盖一个缓存条目，例如：

```
>>> from django.core.cache import cache
>>> from django.core.cache.utils import make_template_fragment_key
# {% cache 500 sidebar username %} 的缓存键
>>> key = make_template_fragment_key('sidebar', [username])
>>> cache.delete(key) # 删除缓存的模板片段
```

16.5 低层缓存 API

有时，缓存渲染的整个页面没有实际意义，而且有点过。你的网站中可能有这么一个视图，它的结果由多个耗资源的查询得到，而且会随着时间而变。此时就不适合使用整站或视图层缓存策略缓存整个页面，因为不能缓存整个结果（部分数据经常变动），但是仍想缓存很少变化的结果。

为此，Django 提供了简单的低层缓存 API。使用这个 API 可以按照任何你想要的粒度把对象存入缓存。任何能安全序列化的 Python 对象都可以缓存，例如字符串、字典、模型对象列表，等等。（多数常见的 Python 对象都能序列化，详情参见 Python 文档。）

16.5.1 访问缓存

CACHES 设置中配置的缓存通过类似字典的对象访问——django.core.cache.caches。在同一个线程中重复请求相同的别名得到的是相同的对象。

```
>>> from django.core.cache import caches
```

```
>>> cache1 = caches['myalias']
>>> cache2 = caches['myalias']
>>> cache1 is cache2
True
```

指定的键不存在时，抛出 InvalidCacheBackendError。为了保证线程安全，每个线程使用不同的缓存后端实例。

默认的缓存可以使用简洁的 django.core.cache.cache 访问：

```
>>> from django.core.cache import cache
```

这个对象等同于 caches['default']。

16.5.2 基本用法

基本的接口是 set(key, value, timeout) 和 get(key)：

```
>>> cache.set('my_key', 'hello, world!', 30)
>>> cache.get('my_key')
'hello, world!'
```

timeout 参数是可选的，默认等于 CACHES 设置中相应后端的 timeout 选项（参见前文）。这个参数的作用是设定值在缓存中存储的秒数。设为 None 时，永久缓存；设为 0 时，不缓存。如果缓存中没有对象，cache.get() 返回 None。

```
# 等 30 秒，让 my_key 过期……

>>> cache.get('my_key')
None
```

不建议在缓存中存储字面值 None，因为返回 None 时无法区分是存储的 None 值，还是由于缓存缺失而导致的返回值。可以在 cache.get() 中指定一个默认值，即缓存中没有对象时返回的值。

```
>>> cache.get('my_key', 'has expired')
'has expired'
```

如果只想在键不存在时添加键，使用 add() 方法。它的参数与 set() 一样，但是如果指定的键已经存在，不会更新缓存。

```
>>> cache.set('add_key', 'Initial value')
>>> cache.add('add_key', 'New value')
>>> cache.get('add_key')
'Initial value'
```

为了确认 add() 是否把值存储到缓存中了，可以检查它的返回值。如果存储了值，add() 返回 True，否则返回 False。此外，还有个 get_many() 接口，它只访问缓存一次。get_many() 返回一个字典，包含你查询的各个键中确实存在而且没有过期的那部分。

```
>>> cache.set('a', 1)
>>> cache.set('b', 2)
>>> cache.set('c', 3)
>>> cache.get_many(['a', 'b', 'c'])
{'a': 1, 'b': 2, 'c': 3}
```

若想高效设定多个值，使用 set_many()，它的参数是键值对构成的字典：

```
>>> cache.set_many({'a': 1, 'b': 2, 'c': 3})
>>> cache.get_many(['a', 'b', 'c'])
{'a': 1, 'b': 2, 'c': 3}
```

与 cache.set() 一样，set_many() 有个可选的 timeout 参数。

键使用 delete() 删除。从缓存中删除特定对象的简易方式如下：

```
>>> cache.delete('a')
```

若想一次删除多个键，使用 delete_many()，它的参数是要删除的键列表：

```
>>> cache.delete_many(['a', 'b', 'c'])
```

最后，若想删除缓存中的全部键，使用 cache.clear()。注意，clear() 将删除缓存中的一切，而不只是由应用程序设定的键。

```
>>> cache.clear()
```

现存的键可以分别使用 incr() 和 decr() 递增和递减。默认，递增或递减的量是 1。如果想使用其他量，为 incr() 和 decr() 提供第二个参数。

如果尝试递增或递减不存在的缓存键，抛出 ValueError。

```
>>> cache.set('num', 1)
>>> cache.incr('num')
2
>>> cache.incr('num', 10)
12
>>> cache.decr('num')
11
>>> cache.decr('num', 5)
6
```

可以使用 close()（前提是缓存后端实现了）关闭与缓存的连接。

```
>>> cache.close()
```

注意，对没有实现 close() 方法的缓存后端来说，调用它没有实际作用。

16.5.3 缓存键的前缀

如果在多个服务器之间共享缓存实例，或者在生产环境和开发环境之间共享，一个服务器缓存的数据可能会被另一个服务器使用。因而，如果不同服务器缓存的数据格式没有区别，可能导致特别难以诊断的问题。

为了避免这种问题，Django 支持为一个服务器的所有缓存键添加前缀。保存或读取缓存键时，Django 会自动在前面加上 KEY_PREFIX 设置设定的值。为不同的 Django 实例设定不同的 KEY_PREFIX，能确保缓存值之间没有冲突。

16.5.4 缓存的版本

修改使用缓存的线上代码后，可能需要清除全部现有的缓存值。这个操作最简单的做法是把整个缓存删掉，但是仍然有效、仍然有用的缓存值也丢失了。为了甄别要删除的缓存值，Django 提供了更好的方法。

Django 的缓存框架有个系统全局版本标识符，由缓存设置中的 VERSION 选项指定。Django 结合前缀、用户提供的缓存键和这个设置的值获取最终的缓存键。

默认情况下，对键的请求自动包含默认的缓存版本号。然而，主要的缓存函数都有个 version 参数，用于指定想设定或读取的缓存键版本。例如：

```
# 设定缓存键的第 2 版
>>> cache.set('my_key', 'hello world!', version=2)
# 获取默认版本（假定版本为 1）
>>> cache.get('my_key')
None
# 获取这个键的第 2 版
>>> cache.get('my_key', version=2)
'hello world!'
```

键的版本号可以分别使用 incr_version() 和 decr_version() 递增和递减。这样可以把指定的键升到新版本，而其他键则不受影响。接着上例：

```
# 递增 my_key 的版本号
>>> cache.incr_version('my_key')
# 默认版本依然不存在
>>> cache.get('my_key')
None
# 第 2 版也不存在了
>>> cache.get('my_key', version=2)
None
# 但是第 3 版存在
>>> cache.get('my_key', version=3)
'hello world!'
```

16.5.5 缓存键的组合方式

前文说过，用户提供的缓存键并不是最终使用的值，还要加上前缀和版本号。默认情况下，这三部分使用冒号连接在一起，组成最终的字符串：

```
def make_key(key, key_prefix, version):
    return ':'.join([key_prefix, str(version), key])
```

如果想以其他方式组合这三部分，或者相对最终得到的键做其他处理（例如计算键的哈希摘要），可以自定义生成键的函数。CACHES 设置中的 KEY_FUNCTION 选项用于指定一个函数的点分路径，函数的原型如上所示。如果提供这个选项，将使用指定的函数代替默认的键组合函数。

16.5.6 缓存键相关的警告

生产环境最常使用的缓存后端 Memcached 不允许缓存键的长度超过 250 个字符，也不允许包含空白或控制字符。如若不然，会导致异常抛出。为了促进不同后端之间的兼容性，尽量减少意外情况，内置的其他几个缓存后端遇到能导致 Memcached 出错的键时会发出警告（django.core.cache.backends.base.CacheKeyWarning）。

如果你在生产环境中使用的后端（自定义的后端，或者是 Memcached 之外的内置后端）能接受更大范围的键，而且不想看到这样的警告，可以在 INSTALLED_APPS 中的某个应用的 management 模块里使用下述代码静默 CacheKeyWarning：

```
import warnings

from django.core.cache import CacheKeyWarning

warnings.simplefilter("ignore", CacheKeyWarning)
```

如果想自定义内置后端验证键的逻辑，可以定义后端的子类，只覆盖 `validate_key` 方法，然后按照 16.1.6 节的说明做。

例如，对本地内存后端来说，在一个模块中编写下述代码：

```
from django.core.cache.backends.locmem import LocMemCache

class CustomLocMemCache(LocMemCache):
    def validate_key(self, key):
        # 自定义验证逻辑，根据需要抛出异常或发出警告
        # ...
```

然后，在 `CACHES` 设置中把 `BACKEND` 设为这个类的点分 Python 路径。

16.6 下游缓存

目前，本章集中讨论的是如何缓存自己的数据。但是，与 Web 开发有关的还有一种缓存：由下游缓存所做的缓存。这是系统做的缓存，发生在请求到达网站之前。下面是几个下游缓存的例子：

- ISP 可能缓存某些页面，当你访问页面时（如 `http://example.com/`），ISP 直接返回缓存的页面，根本不去访问 `http://example.com/`。`http://example.com/` 的维护人员对这种缓存一无所知；ISP 位于网站和 Web 浏览器之间，悄无声息地处理所有缓存。
- Django 网站可能在*代理缓存*（proxy cache）后面，例如 Squid Web Proxy Cache，为了提升性能，代理缓存会缓存页面。此时，请求首先由代理处理，只在需要时才发给你的应用程序。
- Web 浏览器也会缓存页面。如果网页发送适当的首部，后续对那个页面的请求会使用浏览器本地缓存的副本，甚至不会询问网页有没有变化。

下游缓存能在一定程度上提高效率，但是也有危害：很多网页的内容根据通过身份验证与否，以及一堆其他变量而变，假若缓存系统盲目地只基于 URL 保存页面，可能导致后续访问者看到不正确或敏感的数据。

以 Web 电子邮件系统为例。显然，收件箱的内容取决于所登录的用户。如果 ISP 盲目地缓存，后面的访问者就能看到之前登录用户的收件箱。这可不好。

幸好，HTTP 为这个问题提供了解决方案。有几个 HTTP 首部用于指示下游缓存，根据指定变量区分缓存的内容，以及不缓存特定的页面。接下来的几节介绍这些首部。

16.7 使用 `Vary` 首部

`Vary` 首部定义缓存机制构建缓存键时要考虑的请求首部。例如，如果一个网页的内容取决于用户的语言偏好设置，那个页面就在语言上有区别。Django 的缓存系统默认使用所请求页面的完全限定 URL（如 `http://www.example.com/stories/2005/?order_by=author`）创建缓存键。

这意味着，对那个 URL 的所有请求都使用同一个缓存版本，而不管用户代理的区别，例如 cookie 或语言偏好设置。然而，如果页面根据请求首部（如 cookie、语言或用户代理）的不同而生成不同的内容，就要通过 `Vary` 首部告诉缓存机制页面的内容由这些首部而定。

为此，在 Django 中使用便利的 django.views.decorators.vary.vary_on_headers() 视图装饰器，如下所示：

```
from django.views.decorators.vary import vary_on_headers

@vary_on_headers('User-Agent')
def my_view(request):
    # ...
```

此时，缓存系统（例如 Django 的缓存中间件）会针对不同的用户代理创建单独的缓存。与自己动手设定 Vary 首部（例如 response['Vary'] = 'user-agent'）相比，使用 vary_on_headers 装饰器的好处是，它把内容添加到 Vary 首部中（可能已有内容），而不是全新设定，导致现有内容被覆盖。可以把多个首部传给 vary_on_headers()：

```
@vary_on_headers('User-Agent', 'Cookie')
def my_view(request):
    # ...
```

这样设置的作用是，告诉下游缓存，每个用户代理和 cookie 的组合都有自己单独的缓存值。例如，用户代理为“Mozilla”、cookie 值为“foo=bar”的请求，与用户代理为“Mozilla”，而 cookie 值为“foo=ham”的请求是不同的。因为经常会区别 cookie，所以 Django 提供了 django.views.decorators.vary.vary_on_cookie() 装饰器。下面两个视图是等效的：

```
@vary_on_cookie
def my_view(request):
    # ...

@vary_on_headers('Cookie')
def my_view(request):
    # ...
```

传给 vary_on_headers 的首部不区分大小写，因此 'User-Agent' 与 'user-agent' 指的是同一个首部。此外，还可以直接使用辅助函数 django.utils.cache.patch_vary_headers()。这个函数用于设定或把内容添加到 Vary 首部中。例如：

```
from django.utils.cache import patch_vary_headers

def my_view(request):
    # ...
    response = render_to_response('template_name', context)
    patch_vary_headers(response, ['Cookie'])
    return response
```

patch_vary_headers 函数的第一个参数是一个 HttpResponse 实例，第二个参数是由首部名称（不区分大小写）构成的列表或元组。Vary 首部的详细信息参见官方规范。

16.8 使用其他首部控制缓存

与缓存有关的其他问题还有数据隐私和数据存储在层叠缓存的什么位置。通常，用户面对的有两种缓存：浏览器的缓存（私有缓存）和提供商的缓存（公开缓存）。

公开缓存供多人使用，由别人控制。对敏感数据来说，这会导致一些问题，因为你不想让自己的银行账号存储在公开缓存中。所以，Web 应用要通过一种方式告诉缓存哪些数据是私有的，哪些是公开的。

为此，含有私有数据的页面要告知缓存。在 Django 中，使用的是 `cache_control` 视图装饰器。例如：

```
from django.views.decorators.cache import cache_control

@cache_control(private=True)
def my_view(request):
    # ...
```

这个装饰器在背后会发送合适的 HTTP 首部。注意，在这里设定的 `private` 和 `public` 是互斥的。如果应该设为私有的，它会把公开指令删除（反之亦然）。

在同时具有私有文章和公开文章的博客中就可以使用这两个指令。公开的文章可以在共享的缓存中缓存。下述示例使用 `django.utils.cache.patch_cache_control()` 函数（`cache_control` 装饰器在内部调用它）手动修改缓存控制首部：

```
from django.views.decorators.cache import patch_cache_control
from django.views.decorators.vary import vary_on_cookie

@vary_on_cookie
def list_blog_entries_view(request):
    if request.user.is_anonymous():
        response = render_only_public_entries()
        patch_cache_control(response, public=True)
    else:
        response = render_private_and_public_entries(request.user)
        patch_cache_control(response, private=True)

    return response
```

控制缓存参数还有其他方式。例如，HTTP 允许应用程序这么做：

- 定义页面缓存的最长时间。
- 指定是否检查新版，仅在没有变化时使用缓存的内容。（有些缓存即使发现页面内容有变仍然发送缓存的内容，只是因为缓存的副本尚未过期。）

在 Django 中，这些缓存参数使用 `cache_control` 视图装饰器指定。在下面的示例中，`cache_control` 告诉缓存，每次访问都重新验证缓存，而且至少存储缓存的版本 3600 秒：

```
from django.views.decorators.cache import cache_control

@cache_control(must_revalidate=True, max_age=3600)
def my_view(request):
    # ...
```

有效的 `Cache-Control` 首部指令都能传给 `cache_control()` 装饰器。完整的指令列表如下：

- `public=True`
- `private=True`
- `no_cache=True`
- `no_transform=True`
- `must_revalidate=True`

- proxy_revalidate=True
- max_age=num_seconds
- s_maxage=num_seconds

Cache-Control 指令的完整说明参见规范。（注意，缓存中间件已经把首部的 max-age 指令设为 CACHE_MIDDLEWARE_SECONDS 设置的值。cache_control 装饰器中指定的 max_age 参数优先采用，而且能正确合并到 Cache-Control 首部中。）

如果想设定彻底禁止缓存的首部，使用 django.views.decorators.cache.never_cache 视图装饰器。这个装饰器添加的首部确保响应不会被浏览器或其他设备缓存。例如：

```
from django.views.decorators.cache import never_cache

@never_cache
def myview(request):
    # ...
```

16.9 接下来

下一章探讨 Django 的中间件。

第 17 章 Django 中间件

中间件是插在 Django 的请求和响应过程之中的框架。这是一种轻量级的低层插件系统，用于全局调整 Django 的输入或输出。

一个中间件组件专注于做一件特定的事。例如，Django 使用 `AuthenticationMiddleware` 这个中间件组件处理带会话的请求。

本章说明中间件的工作方式、如何激活中间件，以及如何自己动手编写中间件。Django 自带了一些中间件，拿来即用。参见 17.4 节。

17.1 激活中间件

若想激活一个中间件组件，把它添加到 Django 设置文件中的 `MIDDLEWARE_CLASSES` 列表里。

`MIDDLEWARE_CLASSES` 设置中的各个中间件组件使用字符串表示，这个字符串是中间件类名的完整 Python 路径。例如，下面是使用 `django-admin startproject` 命令创建项目后得到的默认值：

```
MIDDLEWARE_CLASSES = [
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.middleware.common.CommonMiddleware',
    'django.middleware.csrf.CsrfViewMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'django.contrib.messages.middleware.MessageMiddleware',
    'django.middleware.clickjacking.XFrameOptionsMiddleware',
]
```

没有任何中间件是必须的，如果愿意，`MIDDLEWARE_CLASSES` 列表可以为空，但是强烈建议至少要使用 `CommonMiddleware`。

`MIDDLEWARE_CLASSES` 要按照一定顺序罗列中间件，因为中间件之间可能彼此依赖。例如，`AuthenticationMiddleware` 在会话中存储通过身份验证的用户，因此必须列在 `SessionMiddleware` 后面。常见 Django 中间件类的顺序参见 17.5 节。

17.2 钩子和应用中间件的顺序

处理请求时，在调用视图之前，Django 按照 `MIDDLEWARE_CLASSES` 列出的顺序从上到下应用各个中间件。这期间有两个钩子可用：

- `process_request()`
- `process_view()`

处理响应时，在调用视图之后，Django 按照相反的顺序从下到上应用各个中间件。这期间有三个钩子可用：

- `process_exception()`
- `process_template_response()`

- process_response()

你可以把这种结构想象成洋葱，外层中间件类包裹着内层视图。下面分述各个钩子。

17.3 自己动手编写中间件

自己编写中间件不难。一个中间件组件就是一个 Python 类，其中定义一个或多个下述方法。

17.3.1 process_request

方法签名：process_request(request)

- request 是一个 HttpRequest 对象。
- 处理请求时，Django 在决定执行哪个视图之前调用 process_request()。

这个方法应该返回 None 或一个 HttpResponse 对象。如果返回 None，Django 继续处理请求，调用其他中间件中的 process_request() 方法，然后调用 process_view() 方法，最后执行恰当的视图。

如果返回一个 HttpResponse 对象，Django 不再调用其他处理请求、视图或异常的中间件，也不执行视图，而是应用响应中间件，返回结果。

17.3.2 process_view

方法签名：process_view(request, view_func, view_args, view_kwargs)

- request 是一个 HttpRequest 对象。
- view_func 是 Django 将使用的 Python 函数。（是函数对象，而非函数名称的字符串形式。）
- view_args 是要传给视图的位置参数列表。
- view_kwargs 是要传给视图的关键字参数字典。
- view_args 和 view_kwargs 中都不包含视图的第一个参数（request）。

process_view() 方法在 Django 调用视图之前的最后一刻调用，应该返回 None 或一个 HttpResponse 对象。如果返回 None，Django 将继续处理请求，调用其他中间件中的 process_view() 方法，然后执行恰当的视图。

如果返回一个 HttpResponse 对象，Django 不会再调用其他视图或异常中间件，也不执行视图，而是应用响应中间件，返回结果。

> **提示**
>
> 在中间件的 process_request 或 process_view 方法中访问 request.POST 的话，在中间件之后运行的视图无法修改请求的上传处理程序，因此通常不应该这么做。

CsrfViewMiddleware 类算是一个例外，它提供的 csrf_exempt() 和 csrf_protect() 装饰器用于控制视图在何时验证 CSRF。

17.3.3 process_template_response

方法签名：process_template_response(request, response)

- request 是一个 HttpRequest 对象。
- response 是一个 TemplateResponse 对象（或其他等效的对象），由 Django 视图或中间件返回。

process_template_response() 在视图执行完毕后立即调用。如果响应实例有 render() 方法，表明它是 TemplateResponse 或等效的对象。

process_template_response() 方法必须返回一个实现 render 方法的响应对象。为此，可以修改传入的 response 的 response.template_name 和 response.context_data，也可以创建并返回全新的 TemplateResponse 或等效的对象。

无需自己动手渲染响应，响应在所有模板响应中间件执行完毕后自动渲染。

处理响应时，中间件反向运行；在这个过程中会调用 process_template_response()。

17.3.4 process_response

方法签名：process_response(request, response)

- request 是一个 HttpRequest 对象。
- response 是 Django 视图或中间件返回的 HttpResponse 或 StreamingHttpResponse 对象。

所有响应在返回给浏览器之前都会调用 process_response() 方法。这个方法必须返回一个 HttpResponse 或 StreamingHttpResponse 对象。为此，可以修改传入的 response，或者创建并返回全新的 HttpResponse 或 StreamingHttpResponse 对象。

与 process_request() 和 process_view() 方法不同，process_response() 方法始终调用，即便跳过了所在中间件类的 process_request() 和 process_view() 方法也是如此（因为前面的中间件返回的是 HttpResponse 对象）。因此，process_response() 方法不能依赖 process_request() 所做的设置。

最后，记住，处理响应时，中间件反向运行，从下到上。这意味着，MIDDLEWARE_CLASSES 中的最后一个中间件类先执行。

处理流式响应

与 HttpResponse 不同的是，StreamingHttpResponse 没有 content 属性。因而，中间件不能假定所有响应都有 content 属性。如果需要访问内容，必须测试是不是流式响应，然后据此调整行为：

```
if response.streaming:
    response.streaming_content = wrap_streaming_content(response.streaming_content)
else:
    response.content = alter_content(response.content)
```

streaming_content 应该视作非常大的对象，内存中放不下。响应中间件可以使用一个新的生成器包装它，一定不能直接使用。streaming_content 经常像下面这样包装：

```
def wrap_streaming_content(content):
    for chunk in content:
        yield alter_content(chunk)
```

17.3.5 process_exception

方法签名：process_exception(request, exception)

- request 是一个 HttpRequest 对象。
- exception 是一个 Exception 对象，由视图函数抛出。

Django 在视图抛出异常时调用 process_exception() 方法。这个方法应该返回 None 或一个 HttpResponse 对象。如果返回 HttpResponse 对象，会应用模板响应和响应中间件，然后把得到的响应返回给浏览器。否则，使用默认的方式处理异常。

再次强调，处理响应时，中间件反向运行；其中包含 process_exception。如果某个异常中间件返回响应，那个中间件上面的中间件类不再调用。

17.3.6 __init__

多数中间件类不需要定义初始化方法，因为中间件类基本上只需实现各个 process_* 方法。但是，如果需要某种全局状态，可以实现 __init__。不过，要留意几个问题：

1. Django 初始化中间件时不传入任何参数，因此你定义的 __init__ 方法不能有参数。
2. process_* 方法每次请求调用一次，而 __init__ 只在 Web 服务器响应第一个请求时调用一次。

把中间件标记为不使用

有时需要在运行时判断是否应该使用中间件。为此，可以在 __init__ 方法中抛出 django.core.exceptions.MiddlewareNotUsed。此时，Django 会从中间件列表中删除那个中间件；如果 DEBUG 的值为 True，还会在 django.request 日志记录器中记录一条调试消息。

17.3.7 其他指导方针

- 中间件类无需继承任何类。
- 中间件类可以放在 Python 路径中的任何位置。对 Django 来说，只需在 MIDDLEWARE_CLASSES 设置中列出中间件类的路径。
- 如果想找示例，看看 Django 自带的中间件。
- 如果你觉得自己编写的中间件组件对别人可能也有用，贡献到社区中！让我们知道你编写了中间件，我们会考虑要不要把它添加到 Django 中。

17.4 可用的中间件

17.4.1 缓存中间件

django.middleware.cache.UpdateCacheMiddleware 和 django.middleware.cache.FetchFromCacheMiddleware

为全站启用缓存。启用这两个中间件后，Django 驱动的每个页面都最长缓存 CACHE_MIDDLEWARE_SECONDS 所定义的秒数。参见第 16 章。

17.4.2 常用中间件

django.middleware.common.CommonMiddleware

为完美主义者增添几分便利：

- 禁止 DISALLOWED_USER_AGENTS 设置中的用户代理访问。这个设置的值是编译后的正则表达式对象列表。
- 根据 APPEND_SLASH 和 PREPEND_WWW 设置重写 URL。

  APPEND_SLASH 设为 True 时，如果原始 URL 不以斜线结尾，而且在 URL 配置中找不到，就在后面添加一条斜线，构成一个新 URL。如果在 URL 配置中找得到这个新 URL，Django 把请求重定向到新 URL。否则，使用常规方式处理原 URL。例如，如果没有匹配 foo.com/bar 的 URL 模式，但有匹配 foo.com/bar/ 的模式，foo.com/bar 将重定向到 foo.com/bar/。

  PREPEND_WWW 设为 True 时，前面没有“www.”的 URL 将重定向到带“www.”的 URL。

  这两个设置的目的是规范化 URL。这背后的思想是，每个 URL 都应该是唯一的。严格来说，foo.com/bar 这个 URL 与 foo.com/bar/ 是有区别的，搜索引擎索引程序会将其视为不同的 URL，所以最好规范化 URL。
- 根据 USE_ETAGS 设置处理 ETag。USE_ETAGS 设为 True 时，Django 会计算页面内容的 MD5 哈希值，用于设定 Etag 首部；而且还会适时发送 Not Modified 响应。

CommonMiddleware.response_redirect_class

默认为 HttpResponsePermanentRedirect。可以继承 CommonMiddleware 类，然后覆盖这个属性，自定义中间件以何种方式重定向。

django.middleware.common.BrokenLinkEmailsMiddleware

遇到死链时发送邮件通知 MANAGERS。

17.4.3 GZip 中间件

> **提醒**
>
> 安全研究人员近期发现，使用压缩技术（包括 GZipMiddleware）的网站有些漏洞，容易受到攻击。攻击者可以利用这些漏洞攻破 Django 的 CSRF 防线。使用 GZipMiddleware 之前要谨慎考虑，以免受到这些攻击。哪怕有一点迟疑，都不应该使用 GZipMiddleware。详情参见 breachattack.com。

为支持 GZip 压缩的浏览器（所有现代的浏览器）压缩内容。

这个中间件应该放在任何需要读写响应主体的中间件之前，保证在所有读写操作之后压缩。

满足下述任何一个条件时不再压缩内容：

- 主体内容少于 200 字节。
- 响应已经设定了 Content-Encoding 首部。
- （浏览器）发送的 Accept-Encoding 请求首部中不包含 gzip。

可以使用 gzip_page() 装饰器在具体的视图上应用 GZip 压缩。

17.4.4 条件 GET 请求中间件

django.middleware.http.ConditionalGetMiddleware

处理条件 GET 请求。如果响应有 ETag 或 Last-Modified 首部，而且请求有 If-None-Match 或 If-Modified-

Since 首部，把响应替换成 HttpResponseNotModified 对象。

还设定 Date 和 Content-Length 响应首部。

17.4.5 区域设置中间件

django.middleware.locale.LocaleMiddleware

根据请求中的数据选择语言，为用户提供定制内容。参见第 18 章。

LocaleMiddleware.response_redirect_class

默认为 HttpResponseRedirect。可以继承 LocaleMiddleware 类，然后覆盖这个属性，自定义中间件以何种方式重定向。

17.4.6 消息中间件

django.contrib.messages.middleware.MessageMiddleware

提供基于 cookie 和会话的消息支持。参见消息文档。

17.4.7 安全中间件

> **提醒**
>
> 如果部署环境允许，通常最好让前端 Web 服务器负责执行 SecurityMiddleware 提供的功能。这样，不由 Django 伺服的请求（例如静态媒体文件或用户上传的文件）与由 Django 伺服的请求具有相同的保护措施。

django.middleware.security.SecurityMiddleware 为请求-响应循环增加了几项安全措施。SecurityMiddleware 的做法是向浏览器发送几个特殊的首部。每个首部都可以在设置中单独启用或禁用。

HTTP 强制安全传输技术

设置：

- SECURE_HSTS_INCLUDE_SUBDOMAINS
- SECURE_HSTS_SECONDS

如果你的网站只允许通过 HTTPS 访问，可以设定 Strict-Transport-Security 首部，告知现代的浏览器（在指定的时长内）拒绝通过不安全的连接访问你的域名。这样能降低受到非 SSL *中间人攻击*（man-in-the-middle，MITM）的风险。

如果把 SECURE_HSTS_SECONDS 设为非零整数值，SecurityMiddleware 会为所有 HTTPS 响应设定这个首部。

启用 HSTS 时，最好先用小值测试一下，例如 SECURE_HSTS_SECONDS = 3600（一小时）。Web 浏览器发现有 HSTS 首部时，会在指定的时长内拒绝通过非安全的连接（使用 HTTP）访问你的域名。

确认所有静态资源都能通过安全的连接伺服之后（即 HSTS 没有导致任何问题），最好增大这个值（经常设为 31536000 秒，即一年），让不常访问的访客受到保护。

此外，如果把 SECURE_HSTS_INCLUDE_SUBDOMAINS 设为 True，SecurityMiddleware 会在 Strict-Transport-Security 首部中加上 includeSubDomains 标签。推荐这样做（假设所有子域名也都只能通过 HTTPS 访问），否则仍然能通过不安全的连接访问子域名。

> **提醒**
>
> HSTS 策略应用于整个域名，而不只是设定 Strict-Transport-Security 首部的那个 URL。因此，仅当整个域名都通过 HTTPS 伺服时才应该使用 HSTS。严格遵守 HSTS 首部的浏览器遇到过期的、自签名的或无效的 SSL 证书时拒绝用户跳过提醒，不允许继续访问。使用 HSTS 时，要确保证书始终有效！

X-Content-Type-Options: nosniff

设置：

- SECURE_CONTENT_TYPE_NOSNIFF

某些浏览器会尝试猜测静态资源的内容类型，继而覆盖 Content-Type 首部。这样虽然可以帮助未正确配置的服务器显示网站，但是也存在安全隐患。

如果网站允许用户上传文件，恶意用户可以上传精心制作的文件，你以为那是无害的，其实却会被浏览器解释为 HTML 或 JavaScript。

为了防止浏览器猜测内容类型，强制始终使用 Content-Type 首部指定的类型，设定 X-Content-Type-Options: nosniff 首部。SECURE_CONTENT_TYPE_NOSNIFF 设为 True 时，SecurityMiddleware 为所有响应设定这个首部。

注意，多数情况下，用户上传的文件不由 Django 伺服；此时，这个设置帮不上什么忙。例如，如果 MEDIA_URL 直接由前端 Web 服务器（Nginx、Apache，等等）伺服，应该在那里设定这个首部。

另一方面，如果需要核对权限后才能下载文件，而且无法在 Web 服务器中设定这个首部，可以使用这个设置。

X-XSS-Protection

设置：

- SECURE_BROWSER_XSS_FILTER

有些浏览器能屏蔽像是 XSS 攻击的内容。为此，浏览器在页面的 GET 或 POST 参数中查找有没有 JavaScript 内容。如果服务器的响应中重放了 JavaScript，浏览器不会渲染，而是显示一个错误页面。

X-XSS-Protection 首部用于控制这种 XSS 过滤功能。

若想启用浏览器的 XSS 过滤功能，强制始终屏蔽可疑的 XSS 攻击，设定 X-XSS-Protection: 1; mode=block 首部。SECURE_BROWSER_XSS_FILTER 设为 True 时，SecurityMiddleware 为所有响应设定这个首部。

> **提醒**
>
> 浏览器的 XSS 过滤功能是个有用的防御措施，但是不能完全依靠它。它不能侦测到全部 XSS 攻击，而且也不是所有浏览器都支持这个首部。一定要验证所有输入，避免受到 XSS 攻击。

SSL 重定向

设置：

- SECURE_REDIRECT_EXEMPT
- SECURE_SSL_HOST
- SECURE_SSL_REDIRECT

如果你的网站同时支持 HTTP 和 HTTPS，多数用户最终默认用的是不安全的连接。为了增强安全，应该把所有 HTTP 连接重定向到 HTTPS。

如果把 SECURE_SSL_REDIRECT 设为 True，SecurityMiddleware 会把所有 HTTP 连接永久重定向（HTTP 301）到 HTTPS。

出于性能上的考虑，最好在 Django 之外，在负载均衡程序或反向代理服务器（如 Nginx）中重定向。不便配置部署环境时才应该使用 SECURE_SSL_REDIRECT 设置。

如果 SECURE_SSL_HOST 设置不为空值，所有重定向都指向那个主机，而非最初请求的主机。

如果网站中有部分页面要通过 HTTP 访问，不重定向到 HTTPS，可以在 SECURE_REDIRECT_EXEMPT 设置中列出匹配那些 URL 的正则表达式。

如果应用程序部署在负载均衡程序或反向代理服务器之后，而且 Django 好像无法辨识请求是不是安全的，可能需要设定 SECURE_PROXY_SSL_HEADER 首部。

17.4.8 会话中间件

django.contrib.sessions.middleware.SessionMiddleware

启用会话支持。详情参见第 15 章。

17.4.9 网站中间件

django.contrib.sites.middleware.CurrentSiteMiddleware

在每个入站 HttpRequest 对象中添加表示当前网站的 site 属性。详情参见网站文档。

17.4.10 身份验证中间件

django.contrib.auth.middleware 提供三个在身份验证过程中使用的中间件：

- *.AuthenticationMiddleware：在每个入站 HttpRequest 对象中添加表示当前已登录用户的 user 属性。
- *.RemoteUserMiddleware：利用 Web 服务器提供的身份验证。
- *.SessionAuthenticationMiddleware：修改密码后让用户的会话失效。在 MIDDLEWARE_CLASSES 中，这个中间件必须出现在 *.AuthenticationMiddleware 后面。

关于在 Django 中验证用户身份的详细说明，参见第 11 章。

17.4.11 CSRF 防护中间件

django.middleware.csrf.CsrfViewMiddleware

为 POST 表单添加隐藏的表单字段，并且检查请求中有没有正确的值，防范*跨站请求伪造*（Cross Site Request Forgery，CSRF）。关于 CSRF 防护的详细说明，参见第 19 章。

17.4.12 X-FRAME-OPTIONS 中间件

`django.middleware.clickjacking.XFrameOptionsMiddleware`

通过 `X-Frame-Options` 首部防范*点击劫持*（clickjacking）。

17.5 中间件的顺序

表 17-1 简要说明一些 Django 中间件类的顺序。

表 17-1：中间件类的顺序

类	说明
`UpdateCacheMiddleware`	在修改 `Vary` 首部的中间件（`SessionMiddleware`、`GZipMiddleware`、`LocaleMiddleware`）前面。
`GZipMiddleware`	在任何可能修改或使用响应主体的中间件前面。在 `UpdateCacheMiddleware` 后面，因为要修改 `Vary` 首部。
`ConditionalGetMiddleware`	在 `CommonMiddleware` 前面，因为设定 `USE_ETAGS = True` 时，要使用 `Etag` 首部。
`SessionMiddleware`	在 `UpdateCacheMiddleware` 后面，因为要修改 `Vary` 首部。
`LocaleMiddleware`	在前部，`SessionMiddleware`（使用会话数据）和 `CacheMiddleware`（修改 `Vary` 首部）后面。
`CommonMiddleware`	在任何可能修改首部的中间件前面（要计算 Etag）。在 `GZipMiddleware` 后面，因为不为压缩的内容计算 Etag。靠近顶部，因为 `APPEND_SLASH` 或 `PREPEND_WWW` 设为 `True` 时要重定向。
`CsrfViewMiddleware`	在任何假定已经防范了 CSRF 攻击的视图中间件前面。
`AuthenticationMiddleware`	在 `SessionMiddleware` 后面，因为要使用会话存储器。
`MessageMiddleware`	在 `SessionMiddleware` 后面，这样才能使用基于会话的存储器。
`FetchFromCacheMiddleware`	在任何修改 `Vary` 首部的中间件后面，因为创建缓存哈希键时要从那个首部中选一个值。
`FlatpageFallbackMiddleware`	应该放在靠近底部的位置，做最后一搏。
`RedirectFallbackMiddleware`	应该放在靠近底部的位置，做最后一搏。

17.6 接下来

下一章讨论 Django 对国际化的支持。

第 18 章 国际化

Django 最初在美国正中开发，我说的是真的，堪萨斯州劳伦斯市离美国大陆的地理中心不到 40 英里。与多数开源项目一样，Django 社区不断发展，吸引了来自世界各地的人。随着 Django 社区不断多元化，*国际化*（internationalization）和*本地化*（localization）变得越来越重要。

Django 自身已经做了全面国际化，所有字符串都标记为可翻译的，而且还有控制本地化输出（如日期和时间）的设置。Django 自带了超过 50 种本地化文件。如果你的母语不是英语，很有可能 Django 已经为你的语言提供了翻译。

Django 自身使用的国际化框架也可用于本地化你的代码和模板。

因为很多开发者没有正确理解国际化和本地化的真正意义，所以我们先下几个定义。

18.1 定义

18.1.1 国际化

指为任何区域设置提供支持的程序设计过程。这个过程通常由软件开发者处理。国际化包括标记可翻译的文本（例如 UI 元素和错误消息）；抽象日期和时间的显示方式，兼顾不同的区域标准；支持不同的时区；以及在代码中不对用户的区域位置做任何假设。国际化经常缩写为 I18N。（数字 18 表示省略了开头的 I 和末尾的 N 之间的 18 个字母。）

18.1.2 本地化

指把国际化的程序翻译成特定区域的语言。这个过程通常由翻译人员处理。本地化有时缩写为 L10N。

处理语言时还会用到一些其他术语：

区域设置名称

`ll` 形式的语言规范码，或者 `ll_CC` 形式的语言和国家规范码。例如：`it`、`de_AT`、`es`、`pt_BR`。语言部分始终小写，国家部分始终大写。中间以一个下划线分开。

语言代码

表示语言的名称。浏览器在 `Accept-Language` 首部中使用这种格式指定接受的语言。例如：`it`、`de-at`、`es`、`pt-br`。语言代码一般使用小写字母表示，但是 `Accept-Language` 首部不区分大小写。分隔符是一个连字符。

消息文件

表示一种语言的纯文本文件，包含所有可翻译的字符串，以及目标语言的表述。消息文件的扩展名是 `.po`。

翻译字符串

可以翻译的文本串。

格式文件

定义目标区域设置所用数据格式的 Python 模块。

18.2 翻译

若想翻译 Django 项目，要在 Python 代码和模板中添加少量的钩子。这些钩子叫做翻译字符串，其作用是告诉 Django，把文本翻译成终端用户的语言（如果消息文件中有文本的翻译）。你要负责把字符串标记为可翻译的，系统只能翻译它知道如何翻译的字符串。

然后，Django 把翻译字符串提取到一个消息文件中。翻译人员在这个文件中使用目标语言翻译字符串。翻译完毕后还要编译。这个过程使用 GNU `gettext` 工具集。

最后，Django 动态地把 Web 应用翻译成用户选择的语言。

其实，Django 在这个过程中做了两件事：

- 让开发者和模板编写人员指定应用中的哪些地方应该翻译。
- 根据用户的语言偏好设置翻译 Web 应用程序。

Django 的国际化钩子默认启用，这意味着在框架的某些位置要处理国际化，因此有些消耗。如果不使用国际化，应该花两秒钟在设置文件中设定 `USE_I18N = False`。这样，Django 会做些优化，不加载国际化机制，省去一些消耗。还有一个与此无关的设置，`USE_L10N`，它用于控制是否让 Django 本地化格式。

18.3 国际化 Python 代码

18.3.1 标准的翻译函数

翻译字符串使用 `ugettext()` 函数指定。为了写起来方便，可以导入为别名 `_`。

Python 标准库中的 `gettext` 模块在全局命名空间中安装 `_()`，作为 `gettext()` 的别名。在 Django 中不应该这么做，原因如下：

1. `ugettext()` 对国际字符集（Unicode）的支持比 `gettext()` 好。有时，在特定的文件中应该把 `ugettext_lazy()` 作为默认的翻译方法。把 `_()` 排除在全局命名空间之外促使开发者思考哪个才是最恰当的翻译函数。
2. 在 Python 的交互式 shell 和 doctest 中，下划线（_）表示前一个结果。如果注册全局的 `_()` 函数，对此行为有影响。把 `ugettext()` 导入为 `_()` 能避免这个问题。

下述示例把"Welcome to my site."标记为翻译字符串：

```
from django.utils.translation import ugettext as _
from django.http import HttpResponse

def my_view(request):
    output = _("Welcome to my site.")
    return HttpResponse(output)
```

当然，也可以不使用别名。下述示例与之等效：

```
from django.utils.translation import ugettext
```

```python
from django.http import HttpResponse

def my_view(request):
    output = ugettext("Welcome to my site.")
    return HttpResponse(output)
```

通过计算得到的值也能翻译。下述示例与之等效：

```python
def my_view(request):
    words = ['Welcome', 'to', 'my', 'site.']
    output = _(' '.join(words))
    return HttpResponse(output)
```

变量也一样能翻译。下述示例与之等效：

```python
def my_view(request):
    sentence = 'Welcome to my site.'
    output = _(sentence)
    return HttpResponse(output)
```

（像上面这样翻译变量和计算得到的值有个问题：Django 检测翻译字符串的工具 `django-admin makemessages` 无法找到它们。`makemessages` 在后文详述。）

传给 `_()` 或 `ugettext()` 的字符串可以带有占位符，通过 Python 标准的具名字符串内插句法指定。例如：

```python
def my_view(request, m, d):
    output = _('Today is %(month)s %(day)s.') % {'month': m, 'day': d}
    return HttpResponse(output)
```

这样便于针对特定的语言重新排列占位文本。例如，英语翻译可能是“Today is November 26.”，而西班牙语翻译则是“Hoy es 26 de Noviembre.”，即月和日的位置对调了。

鉴于此，有多个参数时应该使用具名内插占位符（如 `%(day)s`），不能使用位置内插占位符（如 `%s` 或 `%d`）。如若不然，翻译时无法重新排列占位文本。

18.3.2 给翻译人员看的注释

如果想对翻译人员解释可翻译的字符串，可以在翻译字符串前面一行添加一个以 `Translators` 开头的注释，例如：

```python
def my_view(request):
    # Translators: This message appears on the home page only
    output = ugettext("Welcome to my site.")
```

在得到的 `.po` 文件中，这个注释会与要翻译的内容放在一起，而且多数翻译工具都能把它显示出来。

下面是前例得到的 `.po` 文件中的相应片段：

```
#. Translators: This message appears on the home page only
# path/to/python/file.py:123
msgid "Welcome to my site."
msgstr ""
```

在模板中也可以这么做，详情参见 18.4.4 节。

18.3.3 把字符串标记为暂时无需翻译

`django.utils.translation.ugettext_noop()` 函数的作用是把字符串标记为翻译字符串，但是暂时不翻译，而是从变量中翻译。

要在系统或用户之间交换的不变字符串（例如数据库中的字符串）应该存储在源语言中，而且应该尽量推迟翻译（例如在呈现给用户时），此时就可以使用这个函数。

18.3.4 复数变形

`django.utils.translation.ungettext()` 函数用于指定消息的复数形式。

`ungettext` 接受三个参数：翻译字符串的单数形式，翻译字符串的复数形式，以及对象的数量。

如果想让 Django 应用程序可以本地化到比英语的复数变形复杂的语言（英语有两种形式，单数是“object”，只要超过一个都用“objects”），要使用这个函数。

例如：

```
from django.utils.translation import ungettext
from django.http import HttpResponse

def hello_world(request, count):
    page = ungettext(
        'there is %(count)d object',
        'there are %(count)d objects',
        count
        ) % {
            'count': count,
        }
    return HttpResponse(page)
```

这里，对象的数量通过 `count` 变量传给翻译字符串。

注意，不同语言的复数变形方式不同，不能始终与“1”比较。下述代码看似完备，但是在某些语言中会得到错误的结果：

```
from django.utils.translation import ungettext
from myapp.models import Report

count = Report.objects.count()
if count == 1:
    name = Report._meta.verbose_name
else:
    name = Report._meta.verbose_name_plural

text = ungettext(
    'There is %(count)d %(name)s available.',
    'There are %(count)d %(name)s available.',
    count
    ) % {
      'count': count,
      'name': name
    }
```

别试图自己实现单复数变形逻辑，肯定会出错的。看看下面这个例子：

```
text = ungettext(
    'There is %(count)d %(name)s object available.',
    'There are %(count)d %(name)s objects available.',
    count
    ) % {
      'count': count,
      'name': Report._meta.verbose_name,
    }
```

使用 ungettext() 时，两个文本串中要内插的变量要使用相同的名称。注意，在上述示例中，两个翻译字符串中使用的都是 name 变量。下述示例除了在某些语言中得不到正确的结果之外，本身就是错的：

```
text = ungettext(
    'There is %(count)d %(name)s available.',
    'There are %(count)d %(plural_name)s available.',
    count
    ) % {
      'count': Report.objects.count(),
      'name': Report._meta.verbose_name,
      'plural_name': Report._meta.verbose_name_plural
    }
```

运行 django-admin compilemessages 命令时会得到下述错误：

```
a format specification for argument 'name', as in 'msgstr[0]', doesn't exist in 'msgid'
```

18.3.5 语境标记

有些单词有不同的意思，例如英语中的“May”，它既指一个月份，也表示一个动词。为了让翻译人员根据语境正确翻译多义词，可以使用 django.utils.translation.pgettext() 函数，有复数变形的字符串则使用 django.utils.translation.npgettext() 函数。这两个函数的第一个参数都是说明语境的字符串。

在得到的 .po 文件中，相同的字符串用语境标记了多少次就会出现多少次（语境在 msgctxt 行里），以便翻译人员为各个语境提供翻译。

例如：

```
from django.utils.translation import pgettext

month = pgettext("month name", "May")
```

或者：

```
from django.db import models
from django.utils.translation import pgettext_lazy

class MyThing(models.Model):
    name = models.CharField(help_text=pgettext_lazy(
        'help text for MyThing model', 'This is the help text'))
```

在 .po 文件中是这样的：

```
msgctxt "month name"
msgid "May"
msgstr ""
```

模板标签 trans 和 blocktrans 也支持语境标记。

18.3.6 惰性翻译

若想惰性翻译字符串，即在访问值时翻译，而不是调用翻译函数时翻译，使用 django.utils.translation 中翻译函数的惰性版本（名称以 lazy 结尾）。

这些函数存储字符串的惰性引用，而不是真正的翻译。在字符串上下文中使用字符串时才会翻译，例如渲染模板时。

这其实就相当于调用模块加载时执行的代码路径里的函数。

定义模型、表单和模型表单时都是如此，因为 Django 以类级属性实现相应的字段。鉴于此，在下述各种情形中一定要使用惰性翻译。

模型字段和关系

例如，要像下面这样翻译 name 字段的说明文本：

```
from django.db import models
from django.utils.translation import ugettext_lazy as _

class MyThing(models.Model):
    name = models.CharField(help_text=_('This is the help text'))
```

可以使用 verbose_name 参数把 ForeignKey 和 ManyToManyField 或 OneToOneField 关系的名称标记为可翻译的：

```
class MyThing(models.Model):
    kind = models.ForeignKey(ThingKind, related_name='kinds',   verbose_name=_('kind'))
```

这里设定的 verbose_name 是小写形式。类似地，关系的详细名称也应该使用小写形式，因为 Django 在需要时会自动把首字母变成大写。

模型的详细名称

建议始终明确为模型的类名设定 verbose_name 和 verbose_name_plural，不要依赖默认以英语为中心的生成方式，因为那样得到的结果有时很可笑。

```
from django.db import models
from django.utils.translation import ugettext_lazy as _

class MyThing(models.Model):
    name = models.CharField(_('name'), help_text=_('This is the help text'))

    class Meta:
        verbose_name = _('my thing')
        verbose_name_plural = _('my things')
```

模型方法的 short_description 属性值

可以通过 short_description 属性为模型方法提供翻译，供 Django 和管理后台使用：

```
from django.db import models
from django.utils.translation import ugettext_lazy as _

class MyThing(models.Model):
    kind = models.ForeignKey(ThingKind, related_name='kinds', verbose_name=_('kind'))

    def is_mouse(self):
        return self.kind.type == MOUSE_TYPE
        is_mouse.short_description = _('Is it a mouse?')
```

18.3.7 处理惰性翻译对象

ugettext_lazy() 调用得到的结果可以在任何能使用 Unicode 字符串（类型为 unicode 的对象）的 Python 代码中使用。但是不能在期待字节字符串（str 对象）的地方使用，因为 ugettext_lazy() 返回的对象不知道怎么把自己转换成字节字符串。此外，也不能在字节字符串中使用 Unicode 字符串，这与 Python 的行为是一致的。例如：

```
# 可以这么做：在 Unicode 字符串中放置一个 Unicode 代理
"Hello %s" % ugettext_lazy("people")

# 不能这么做，因为不能在字节字符串中插入 Unicode 对象
# （也不能放置 Unicode 代理）
b"Hello %s" % ugettext_lazy("people")
```

如果看到类似 "hello <django.utils.functional…>" 这样的输出，说明你把 ugettext_lazy() 的结果插入字节字符串中了，也就是代码有问题。

如果不想输入 ugettext_lazy 这么长的名称，可以像下面这样创建别名 _（下划线）：

```
from django.db import models
from django.utils.translation import ugettext_lazy as _

class MyThing(models.Model):
    name = models.CharField(help_text=_('This is the help text'))
```

经常使用 ugettext_lazy() 和 ungettext_lazy() 标记模型和实用函数中的字符串。在代码中处理这些字符串时要小心，不要将其转换成字符串，因为应该尽量延后转换操作（这样才能让区域设置起作用）。鉴于此，有必要使用接下来说明的辅助函数。

惰性翻译和复数变形

使用惰性翻译函数（[u]n[p]gettext_lazy）处理字符串的复数形式时，在定义字符串之时一般不知道 number 参数的值。此时，允许通过键名指定 number 参数。这样，内插字符串时会在字典的那个键名下查找 number 的值。例如：

```
from django import forms
from django.utils.translation import ugettext_lazy

class MyForm(forms.Form):
```

```
    error_message = ungettext_lazy("You only provided %(num)d
      argument", "You only provided %(num)d arguments", 'num')

    def clean(self):
        # ...
        if error:
            raise forms.ValidationError(self.error_message %
              {'num': number})
```

如果字符串中只有一个匿名占位符，可以直接内插 number 参数：

```
class MyForm(forms.Form):
    error_message = ungettext_lazy("You provided %d argument",
        "You provided %d arguments")

    def clean(self):
        # ...
        if error:
            raise forms.ValidationError(self.error_message % number)
```

连接字符串：string_concat()

在包含惰性翻译对象的列表上不能使用 Python 标准的方式连接字符串（''.join([…])），而要使用 django.utils.translation.string_concat() 函数。这个函数创建一个惰性对象，把内容拼接起来，在字符串中插入结果时才将其转换成字符串。例如：

```
from django.utils.translation import string_concat
from django.utils.translation import ugettext_lazy
# ...
name = ugettext_lazy('John Lennon')
instrument = ugettext_lazy('guitar')
result = string_concat(name, ': ', instrument)
```

这里，仅当在字符串中使用 result 时（通常是在渲染模板时）才把里面的惰性翻译转换成字符串。

在其他想延迟翻译的地方使用惰性翻译字符串

在其他地方，如果想延迟翻译，但是要把可翻译的字符串作为参数传给别的函数，可以使用 lazy 包装。例如：

```
from django.utils import six  # 兼容 Python 3
from django.utils.functional import lazy
from django.utils.safestring import mark_safe
from django.utils.translation import ugettext_lazy as _

mark_safe_lazy = lazy(mark_safe, six.text_type)
```

然后这么用：

```
lazy_string = mark_safe_lazy(_("<p>My <strong>string!</strong></p>"))
```

18.3.8 语言的本地化名称

`get_language_info()` 函数提供关于语言的详细信息：

```
>>> from django.utils.translation import get_language_info
>>> li = get_language_info('de')
>>> print(li['name'], li['name_local'], li['bidi'])
German Deutsch False
```

`name` 和 `name_local` 键的值分别是语言的英文名和在目标语言中的名称。仅当是双向书写的语言时，`bidi` 键的值才为 `True`。

语言信息在 `django.conf.locale` 模块中。在模板代码中也能以类似的方式访问这些信息。详情参见下文。

18.4 国际化模板代码

Django 模板使用两个模板标签翻译，而且与在 Python 代码中的句法稍有不同。若想在模板中访问那两个标签，在模板顶部写上 `{% load i18n %}`。与其他模板标签一样，每个需要翻译的模板都要有这么一个 `load` 标签，即便是扩展自已经加载了 `i18n` 标签的模板也是如此。

18.4.1 trans 模板标签

`{% trans %}` 模板标签既能用于翻译不变的字符串（放在单引号或双引号中），也能用于翻译变量的内容：

```
<title>{% trans "This is the title." %}</title>
<title>{% trans myvar %}</title>
```

如果指定 `noop` 选项，会查找变量，但不翻译。这么做只是为了占位，而把翻译留到未来某个时刻：

```
<title>{% trans "myvar" noop %}</title>
```

行间翻译在背后调用 `ugettext()` 函数。

如果传给 `trans` 标签的是一个模板变量，在运行时首先把变量解析成字符串，然后在消息目录中查找那个字符串。

在 `{% trans %}` 标签中，不能内插模板变量。如果需要变量中的字符串（占位符），使用 `{% blocktrans %}`。如果想获取翻译后的字符串，但不将其显示出来，使用下述句法：

```
{% trans "This is the title" as the_title %}
```

在实际运用中，如果需要在多个地方使用翻译后的字符串，或者作为参数传给别的模板标签或过滤器，就可以这么做：

```
{% trans "starting point" as start %}
{% trans "end point" as end %}
{% trans "La Grande Boucle" as race %}

<h1>
  <a href="/" title="{% blocktrans %}Back to '{{ race }}' homepage{% endblocktrans %}">{{ race
}}</a>
</h1>
<p>
{% for stage in tour_stages %}
```

```
    {% cycle start end %}: {{ stage }}{% if forloop.counter|divisibleby:2 %}<br />{% else %},
{% endif %}
{% endfor %}
</p>
```

{% trans %} 标签还支持使用 context 关键字指定语境标记：

```
{% trans "May" context "month name" %}
```

18.4.2 blocktrans 模板标签

blocktrans 标签用于标记复杂的句子，包含字面量和变量：

```
{% blocktrans %}This string will have {{ value }} inside.{% endblocktrans %}
```

表达式，如对象的属性或模板过滤器，要绑定到变量上，然后在 blocktrans 块中使用那个变量。例如：

```
{% blocktrans with amount=article.price %}
That will cost $ {{ amount }}.
{% endblocktrans %}

{% blocktrans with myvar=value|filter %}
This will have {{ myvar }} inside.
{% endblocktrans %}
```

一个 blocktrans 标签中可以使用多个表达式：

```
{% blocktrans with book_t=book|title author_t=author|title %}
This is {{ book_t }} by {{ author_t }}
{% endblocktrans %}
```

这个示例也可以写成：

```
{% blocktrans with book|title as book_t and author|title as author_t %}
```

在 blocktrans 标签中不能使用其他块级标签（如 {% for %} 或 {% if %}）。

如果无法解析传入的参数，blocktrans 使用 deactivate_all() 函数临时停用当前语言，回落到默认语言。

这个标签也支持复数变形。使用方法如下：

- 使用 count 指定并绑定一个数值。这个数值用于选择正确的单复数形式。
- 在 {% blocktrans %} 和 {% endblocktrans %} 之间以 {% plural %} 标签分隔单数形式和复数形式。

举个例子：

```
{% blocktrans count counter=list|length %}
There is only one {{ name }} object.
{% plural %}
There are {{ counter }} {{ name }} objects.
{% endblocktrans %}
```

一个更复杂的例子：

```
{% blocktrans with amount=article.price count years=i.length %}
```

```
That will cost $ {{ amount }} per year.
{% plural %}
That will cost $ {{ amount }} per {{ years }} years.
{% endblocktrans %}
```

如果在使用计数值的同时要做单复数变形，还要把值绑定到局部变量上，blocktrans 结构在背后会转换成 ungettext 调用。这意味着，在 ungettext 中使用变量的规则在这里也适用。

在 blocktrans 块中无法反向查找 URL，因此应该事先查找好（并存储起来）：

```
{% url 'path.to.view' arg arg2 as the_url %}
{% blocktrans %}
This is a URL: {{ the_url }}
{% endblocktrans %}
```

{% blocktrans %} 还支持通过 context 关键字指定语境：

```
{% blocktrans with name=user.username context "greeting" %}
Hi {{ name }}
{% endblocktrans %}
```

{% blocktrans %} 还支持 trimmed 选项，把标签内容首尾的换行符去掉，把每一行首尾的多个空白替换成一个空格，把各行合并成一行。

这样在 {% blocktrans %} 标签中就可以缩进内容，而不必担心缩进字符会出现在 PO 文件中，免得妨碍翻译。

例如，对下述 {% blocktrans %} 标签来说：

```
{% blocktrans trimmed %}
  First sentence.
  Second paragraph.
{% endblocktrans %}
```

在得到的 PO 文件中是 "First sentence. Second paragraph."，而不是未指定 trimmed 选项时的 "\n First sentence.\n Second sentence.\n"。

18.4.3 传给标签和过滤器的字符串字面量

可以使用熟悉的 _() 句法翻译作为参数传给标签和过滤器的字符串字面量：

```
{% some_tag _("Page not found") value|yesno:_("yes,no") %}
```

这样，标签和过滤器得到的都是翻译后的字符串，无需关心如何翻译。

这里，传给翻译框架的是字符串 "yes,no"，而不是单独的 "yes" 和 "no"。翻译后的字符串要保留逗号，这样过滤器解析代码才知道如何把参数拆开。例如，德语翻译人员可以把字符串 "yes,no" 翻译成 "ja,nein"（保留逗号）。

18.4.4 模板中给翻译人员看的注释

与在 Python 代码中一样，给翻译人员看的说明使用注释编写。既可以使用 comment 标签：

```
{% comment %}Translators: View verb{% endcomment %}
```

```
{% trans "View" %}

{% comment %}Translators: Short intro blurb{% endcomment %}
<p>{% blocktrans %}
    A multiline translatable literal.
   {% endblocktrans %}
</p>
```

也可以使用一行形式的注释结构 `{# … #}`：

```
{# Translators: Label of a button that triggers search #}
<button type="submit">{% trans "Go" %}</button>

{# Translators: This is a text of the base template #}
{% blocktrans %}Ambiguous translatable block of text{% endblocktrans %}
```

下面是前例得到的 .po 文件中的相应片段：

```
#. Translators: View verb
# path/to/template/file.html:10
msgid "View"
msgstr ""

#. Translators: Short intro blurb
# path/to/template/file.html:13
msgid ""
"A multiline translatable"
"literal."
msgstr ""

# ...

#. Translators: Label of a button that triggers search
# path/to/template/file.html:100
msgid "Go"
msgstr ""

#. Translators: This is a text of the base template
# path/to/template/file.html:103
msgid "Ambiguous translatable block of text"
msgstr ""
```

18.4.5 在模板中切换语言

若想在模板中选择语言，使用 `language` 模板标签：

```
{% load i18n %}

{% get_current_language as LANGUAGE_CODE %}
<!-- Current language: {{ LANGUAGE_CODE }} -->
<p>{% trans "Welcome to our page" %}</p>

{% language 'en' %}
```

```
        {% get_current_language as LANGUAGE_CODE %}
        <!-- Current language: {{ LANGUAGE_CODE }} -->
        <p>{% trans "Welcome to our page" %}</p>

    {% endlanguage %}
```

第一处“Welcome to our page”使用当前语言，但是第二处始终使用英语。

18.4.6 其他标签

这些标签也需要 `{% load i18n %}`。

- `{% get_available_languages as LANGUAGES %}` 返回一个元组列表，元组的第一个元素是语言代码，第二个元素是语言名称（翻译成当前启用的语言）。
- `{% get_current_language as LANGUAGE_CODE %}` 返回一个字符串，表示当前用户选择的语言。例如：`en-us`。（参见 18.10.2 节。）
- `{% get_current_language_bidi as LANGUAGE_BIDI %}` 返回当前区域设置的书写方向。为 `True` 时，表示由右向左书写的语言，如希伯来语和阿拉伯语；为 `False` 时，表示由左向右书写的语言，如英语、法语、德语，等等。

启用 `django.template.context_processors.i18n` 上下文处理器后，每个 `RequestContext` 对象都能访问 `LANGUAGES`、`LANGUAGE_CODE` 和 `LANGUAGE_BIDI`（相应的说明参见前文）。

新项目默认未启用 `i18n` 上下文处理器。

使用相应的模板标签和过滤器还能获取任何一个可用语言的信息。若想获得某个语言的信息，使用 `{% get_language_info %}` 标签：

```
{% get_language_info for LANGUAGE_CODE as lang %}
{% get_language_info for "pl" as lang %}
```

然后可以访问下述信息：

```
Language code: {{ lang.code }}<br />
Name of language: {{ lang.name_local }}<br />
Name in English: {{ lang.name }}<br />
Bi-directional: {{ lang.bidi }}
```

此外，还可以使用 `{% get_language_info_list %}` 标签获得一组语言（例如存储在 `LANGUAGES` 中的可用语言）的信息。文档中有个示例，说明如何使用 `{% get_language_info_list %}` 显示一个语言选择菜单。

除了 `LANGUAGES` 那种元组列表之外，`{% get_language_info_list %}` 也支持由语言代码构成的简单列表。假如视图中有下属代码：

```
context = {'available_languages': ['en', 'es', 'fr']}
return render(request, 'mytemplate.html', context)
```

就可以在模板中迭代各个语言：

```
{% get_language_info_list for available_languages as langs %}
{% for lang in langs %} ... {% endfor %}
```

为了方便，还提供了相应的过滤器：

- `{{ LANGUAGE_CODE|language_name }}` (German)
- `{{ LANGUAGE_CODE|language_name_local }}` (Deutsch)
- `{{ LANGUAGE_CODE|language_bidi }}` (False)

18.5 国际化 JavaScript 代码

为 JavaScript 添加翻译面临诸多问题：

- JavaScript 代码无法访问 `gettext`。
- JavaScript 代码无法访问 `.po` 或 `.mo` 文件；它们要由服务器分发。
- JavaScript 的翻译目录要尽量保持小身材。

为了解决这些问题，Django 提供了一种集成方案：把翻译传入 JavaScript，因此可以在 JavaScript 代码中访问 `gettext` 等。

18.5.1 javascript_catalog 视图

这些问题的主要解决方案是 `django.views.i18n.javascript_catalog()` 视图，它发送一个 JavaScript 代码库，里面包含模拟 `gettext` 接口的函数和翻译字符串数组。

这些翻译字符串来自应用程序或 Django 核心，具体内容取决于在 `info_dict` 或 URL 中作何指定。`LOCALE_PATHS` 设置列出的路径也包含在内。

这个视图的用法如下：

```
from django.views.i18n import javascript_catalog

js_info_dict = {
    'packages': ('your.app.package',),
}

urlpatterns = [
    url(r'^jsi18n/$', javascript_catalog, js_info_dict),
]
```

`packages` 中的各个字符串应该是点分 Python 路径（与 `INSTALLED_APPS` 中的字符串格式一样），指向包含 `locale` 目录的包。如果指定多个包，各个包中的翻译目录会合并成一个。这样，在 JavaScript 中就可以使用不同应用中的翻译字符串。

翻译字符串的使用顺序是，`packages` 参数中靠后的包比靠前的包优先级高。遇到翻译冲突时要知道这一点。

这个视图默认使用 gettext 的 `djangojs` 域，不过可以使用 `domain` 参数修改。

可以把包名放入 URL 模式，让这个视图动态加载翻译：

```
urlpatterns = [
    url(r'^jsi18n/(?P<packages>\S+?)/$', javascript_catalog),
]
```

此时，可以在 URL 中指定包名列表（以加号分隔）。如果一个页面使用不同应用中的代码，而且所用的应用经常变动，就可以这么做，因为你并不想加载一个大的目录文件。出于安全考虑，在 URL 中指定的值要么是

django.conf，要么是 INSTALLED_APPS 设置中的某个包。

在 LOCALE_PATHS 设置所列路径里找到的 JavaScript 翻译始终包含在内。为了让翻译查找顺序算法与 Python 和模板保持一致，LOCALE_PATHS 设置中列出的目录，靠前的优先级比靠后的高。

18.5.2 使用 JavaScript 翻译目录

若想使用翻译目录，像下面这样动态生成 script 标签：

```
<script type="text/javascript" src="{% url 'django.views.i18n.javascript_catalog' %}"></script>
```

这行代码通过反向 URL 查找机制找出 JavaScript 目录视图的 URL。加载目录后，JavaScript 代码可以使用标准的 gettext 接口访问翻译目录：

```
document.write(gettext('this is to be translated'));
```

ngettext 接口也可用：

```
var object_cnt = 1 // 或 0、2、3, ...
s = ngettext('literal for the singular case',
      'literal for the plural case', object_cnt);
```

甚至还有字符串插值函数：

```
function interpolate(fmt, obj, named);
```

插值句法借鉴自 Python，因此 interpolate 函数既支持基于位置插值，也支持基于名称插值。

- 基于位置插值：obj 是一个 JavaScript 数组对象，里面的元素依序内插到 fmt 中的占位符所在的位置。例如：

  ```
  fmts = ngettext('There is %s object. Remaining: %s',
           'There are %s objects. Remaining: %s', 11);
  s = interpolate(fmts, [11, 20]);
  // s 的值是 'There are 11 objects. Remaining: 20'
  ```

- 基于名称插值：把 named 参数设为 true 时选择这个模式。obj 是一个 JavaScript 对象或关联数组。例如：

  ```
  d = {
      count: 10,
      total: 50
  };

  fmts = ngettext('Total: %(total)s, there is %(count)s object',
    'there are %(count)s of a total of %(total)s objects', d.count);
  s = interpolate(fmts, d, true);
  ```

不过，别过量使用，这毕竟是 JavaScript 代码，要不断通过正则表达式代换。这种模拟的方式没有 Python 字符串插值的速度快，因此应该留给真正需要的情形（例如结合 ngettext，输出正确的复数形式）。

18.5.3 注意性能

每次请求 javascript_catalog() 视图，它都从 .mo 文件中生成翻译目录。因为输出是不变的（至少对网站的

特定版本来说是如此），所以或许可以缓存。

服务器端缓存能降低 CPU 负载，通过 cache_page() 装饰器可以轻易实现。为了在翻译有变化时让缓存失效，可以提供一个与版本有关的键前缀（如下例所示），或者把这个视图映射到带版本号的 URL 上。

```
from django.views.decorators.cache import cache_page
from django.views.i18n import javascript_catalog

# 翻译有变化时，get_version() 返回的值必须变化
@cache_page(86400, key_prefix='js18n-%s' % get_version())
def cached_javascript_catalog(request, domain='djangojs', packages=None):
    return javascript_catalog(request, domain, packages)
```

客户端缓存能节省带宽，让网站加载的速度变得更快。如果使用 ETag（USE_ETAGS = True），客户端缓存就已经启用了。否则，可以使用条件请求装饰器。在下述示例中，重启应用服务器后缓存便失效。

```
from django.utils import timezone
from django.views.decorators.http import last_modified
from django.views.i18n import javascript_catalog

last_modified_date = timezone.now()

@last_modified(lambda req, **kw: last_modified_date)
def cached_javascript_catalog(request, domain='djangojs', packages=None):
    return javascript_catalog(request, domain, packages)
```

甚至还可以在部署过程中事先生成 JavaScript 翻译目录，然后作为静态文件伺服。

18.6 国际化 URL 模式

Django 为国际化 URL 模式提供了两种机制：

- 在 URL 模式前面添加语言前缀，让 LocaleMiddleware 从所请求的 URL 中检测要使用的语言。
- 通过 django.utils.translation.ugettext_lazy() 函数把 URL 模式本身标记为可以翻译的。

这两种机制都要求为每个请求设定语言；也就是说，MIDDLEWARE_CLASSES 设置中要列出 django.middleware.locale.LocaleMiddleware。

18.6.1 在 URL 模式中添加语言前缀

可以在根 URL 配置中使用这个函数，此时 Django 会自动把当前语言代码添加到 i18n_patterns() 定义的所有 URL 模式前面。例如：

```
from django.conf.urls import include, url
from django.conf.urls.i18n import i18n_patterns
from about import views as about_views
from news import views as news_views
from sitemap.views import sitemap

urlpatterns = [
    url(r'^sitemap\.xml$', sitemap, name='sitemap_xml'),
]
```

```
news_patterns = [
    url(r'^$', news_views.index, name='index'),
    url(r'^category/(?P<slug>[\w-]+)/$',
        news_views.category,
        name='category'),
    url(r'^(?P<slug>[\w-]+)/$', news_views.details, name='detail'),
]

urlpatterns += i18n_patterns(
    url(r'^about/$', about_views.main, name='about'),
    url(r'^news/', include(news_patterns, namespace='news')),
)
```

这样定义 URL 模式后，Django 会自动在 `i18n_patterns` 函数添加的 URL 模式前面加上语言前缀。例如：

```
>>> from django.core.urlresolvers import reverse
>>> from django.utils.translation import activate
>>> activate('en')
>>> reverse('sitemap_xml')
'/sitemap.xml'
>>> reverse('news:index')
'/en/news/'

>>> activate('nl')
>>> reverse('news:detail', kwargs={'slug': 'news-slug'})
'/nl/news/news-slug/'
```

`i18n_patterns()` 只允许在根 URL 配置中使用。如果在引入的 URL 配置中使用，会抛出 `ImproperlyConfigured` 异常。

18.6.2 翻译 URL 模式

URL 模式可以使用 `ugettext_lazy()` 函数标记为可翻译的。例如：

```
from django.conf.urls import include, url
from django.conf.urls.i18n import i18n_patterns
from django.utils.translation import ugettext_lazy as _

from about import views as about_views
from news import views as news_views
from sitemaps.views import sitemap

urlpatterns = [
    url(r'^sitemap\.xml$', sitemap, name='sitemap_xml'),
]

news_patterns = [
    url(r'^$', news_views.index, name='index'),
    url(_(r'^category/(?P<slug>[\w-]+)/$'),
        news_views.category,
        name='category'),
    url(r'^(?P<slug>[\w-]+)/$', news_views.details, name='detail'),
```

```
]

urlpatterns += i18n_patterns(
    url(_(r'^about/$'), about_views.main, name='about'),
    url(_(r'^news/'), include(news_patterns, namespace='news')),
)
```

有了翻译之后，`reverse()` 函数会返回当前语言的 URL。例如：

```
>>> from django.core.urlresolvers import reverse
>>> from django.utils.translation import activate

>>> activate('en')
>>> reverse('news:category', kwargs={'slug': 'recent'})
'/en/news/category/recent/'

>>> activate('nl')
>>> reverse('news:category', kwargs={'slug': 'recent'})
'/nl/nieuws/categorie/recent/'
```

多数情况下，最好只在已经添加语言代码前缀的模式块（使用 `i18n_patterns()`）中使用翻译的 URL，以防翻译的 URL 不小心与未翻译的 URL 模式冲突。

18.6.3 在模板中反向解析

在模板中，反向解析得到的本地化 URL 始终使用当前语言。若想链接到另一个语言，使用 `language` 模板标签。这个标签在所包含的块中启用指定的语言。

```
{% load i18n %}

{% get_available_languages as languages %}

{% trans "View this category in:" %}
{% for lang_code, lang_name in languages %}
    {% language lang_code %}
    <a href="{% url 'category' slug=category.slug %}">{{ lang_name }}</a>
    {% endlanguage %}
{% endfor %}
```

`language` 标签只有一个参数，即语言代码。

18.7 创建本地语言文件

把应用程序中的字符串字面量标记为可翻译的之后，还要翻译。下面说明具体做法。

18.7.1 消息文件

首先，要为新语言创建消息文件。这是一种纯文本文件，针对一门语言，包含所有可翻译的字符串，以及目标语言的表述。消息文件的扩展名为 `.po`。

Django 自带的 `django-admin makemessages` 命令用于自动创建和维护消息文件。

makemessages 命令（以及后文要讨论的 compilemessages 命令）使用 GNU gettext 工具集中的几个命令：xgettext、msgfmt、msgmerge 和 msguniq。

Django 支持的 gettext 最低版本为 0.15。

创建或更新消息文件使用下述命令：

```
django-admin makemessages -l de
```

其中，de 是消息文件的语言名称。例如，巴西葡萄牙语是 pt_BR，奥地利德语是 de_AT，印尼语是 id。

这个脚本应该在下述两个位置运行：

- Django 项目的根目录（manage.py 文件所在的目录）。
- 某个 Django 应用的根目录。

这个脚本扫描项目或应用的源码树，找出所有标记为可翻译的字符串（参见 18.10.3 节，还要确保正确配置了 LOCALE_PATHS），然后在 locale/LANG/LC_MESSAGES 目录中创建（或更新）消息文件。对 de 语言来说，创建的文件是 locale/de/LC_MESSAGES/django.po。

在项目的根目录中运行 makemessages 命令时，提取出来的字符串会自动分配到正确的消息文件中。即，如果应用中有 locale 目录，从那个应用的文件中提取出来的字符串保存到 locale 目录中；否则，提取的字符串保存到 LOCALE_PATHS 设置中的第一个目录里；如果 LOCALE_PATHS 为空，报错。

django-admin makemessages 命令会扫描每个扩展名为 .html 和 .txt 的文件。若想覆盖默认值，使用 --extension 或 -e 选项指定文件扩展名：

```
django-admin makemessages -l de -e txt
```

多个扩展名使用逗号分开，也可以多次使用 --extension 或 -e 选项：

```
django-admin makemessages -l de -e html,txt -e xml
```

> **提醒**
>
> 为 JavaScript 源码创建消息文件时，要使用特殊的域“djangojs”，而非 -e js。

如果没有安装 gettext 实用工具，makemessages 创建的是空文件。遇到这种情况，要么安装 gettext 实用工具，要么复制英语消息文件（如果有的话，locale/en/LC_MESSAGES/django.po），在此基础上翻译。

在 Windows 中，若想使用 makemessages 命令，要安装 GNU gettext 实用工具，详情参见 18.7.4 节。

.po 文件的格式很简单。每个 .po 文件中都有少量的元数据，例如翻译维护人员的联系信息，余下的大部分则是消息列表，即一对对翻译语言和针对目标语言的翻译。

假如 Django 应用中有下面这个翻译字符串：

```
_("Welcome to my site.")
```

django-admin makemessages 命令创建的 .po 文件中会包含下述片段，即一个消息：

```
#: path/to/python/module.py:23
msgid "Welcome to my site."
msgstr ""
```

简单说明一下：

- msgid 是源码中的翻译字符串。不要修改。
- msgstr 是针对目标语言的翻译。一开始是空的，由你负责翻译。记住，翻译要放在引号里。
- 为了便于参考，在每个消息中，msgid 那行上面都有一行注释（以 # 开头），指明翻译字符串所在的文件名和行号。

较长的消息有些特殊。此时，紧跟在 msgstr（或 msgid）后面的是一个空字符串，内容在接下来的几行中，而且一行一个字符串。这些字符串直接拼接在一起。别忘了在每个字符串末尾添加空格，否则两行内容就连在一起了。

考虑到 gettext 工具的内部运作方式，以及要在 Django 的核心和应用程序中使用非 ASCII 字符串，PO 文件的编码必须使用 UTF-8（创建 PO 文件时默认使用这个编码）。这样每个人使用的编码都相同，便于 Django 处理 PO 文件。

若想重新扫描源码和模板，找出新的翻译字符串，更新所有语言的所有消息文件，运行下述命令：

```
django-admin makemessages -a
```

18.7.2 编译消息文件

创建消息文件，以及每次修改消息文件之后，要使用 gettext 把它编译成更高效的格式。这一步通过 django-admin compilemessages 命令操作。

这个命令把所有 .po 文件编译成 .mo 文件，这是一种二进制文件，针对 gettext 做了优化。在运行 django-admin makemessages 命令的那个目录中运行下述命令：

```
django-admin compilemessages
```

就这样。现在翻译可供使用了。

在 Windows 中，若想使用 django-admin compilemessages 命令，要安装 GNU gettext 实用工具，详情参见 18.7.4 节。

Django 只支持 UTF-8 编码的 .po 文件，而且不能有 BOM（Byte Order Mark，*字节序标记*），因此如果你的文本编辑器默认在文件开头添加 BOM，要重新配置编辑器。

18.7.3 为 JavaScript 源码创建消息文件

对 JavaScript 代码来说，创建和更新消息文件的方式与 Django 中的其他消息文件一样，使用 django-admin makemessages 命令。唯一的区别是，要明确把域设为 djangojs，即提供 -d djangojs 参数，如下所示：

```
django-admin makemessages -d djangojs -l de
```

这个命令创建或更新德语的 JavaScript 消息文件。更新消息文件之后，像其他 Django 消息文件一样，运行 django-admin compilemessages 命令。

18.7.4 在 Windows 中安装 gettext

只有想提取消息 ID 或编译消息文件（.po）的人才需要安装 gettext。翻译本身只需要编辑现有的消息文件，但是如果你自己想创建消息文件，或者想测试或编译有改动的消息文件，就要安装 gettext 实用工具。

- 从 GNOME 服务器中下载以下两个 ZIP 文件：
  - gettext-runtime-X.zip
  - gettext-tools-X.zip

  （其中 X 是版本号，必须使用 0.15 或以上版本。）
- 把两个 ZIP 文件中的 bin\ 目录里的内容提取出来，放到系统中的同一个文件夹里（例如 C:\Program Files\gettext-utils）。
- 更新系统路径：
  - 依次打开"控制面板 > 系统 > 高级 > 环境变量"
  - 在"系统变量"列表中点击 Path，再点击"编辑"。
  - 在"变量值"字段后面添加 ;C:\Program Files\gettext-utils\bin。

也可以使用从别处获取的 gettext 二进制文件，只要能正确运行 xgettext --version 就行。如果在 Windows 命令提示符中运行 xgettext --version 命令后弹出一个对话框，提示 xgettext.exe 出错，Windows 将把它关闭，此时千万别使用与 gettext 包有关的 Django 翻译工具。

18.7.5 自定义 makemessages 命令

如果想把额外参数传给 xgettext，要自定义 makemessages 命令，覆盖 xgettext_options 属性：

```
from django.core.management.commands import makemessages

class Command(makemessages.Command):
    xgettext_options = makemessages.Command.xgettext_options + ['--keyword=mytrans']
```

如果想更灵活一些，还可以为自定义的 makemessages 命令增加一个参数：

```
from django.core.management.commands import makemessages

class Command(makemessages.Command):

    def add_arguments(self, parser):
        super(Command, self).add_arguments(parser)
        parser.add_argument('--extra-keyword',
                            dest='xgettext_keywords',
                            action='append')

    def handle(self, *args, **options):
        xgettext_keywords = options.pop('xgettext_keywords')
        if xgettext_keywords:
            self.xgettext_options = (
                makemessages.Command.xgettext_options[:] +
                ['--keyword=%s' % kwd for kwd in xgettext_keywords]
            )
        super(Command, self).handle(*args, **options)
```

18.8 显式设定当前语言

你可能想显式为当前会话设定语言，比如说用户的语言偏好设置从其他系统中获取时。前面已经介绍过 django.utils.translation.activate()。这个函数只影响当前线程。如果想在整个会话中都使用所设的语言，还要修改会话中的 LANGUAGE_SESSION_KEY：

```
from django.utils import translation
user_language = 'fr'
translation.activate(user_language)
request.session[translation.LANGUAGE_SESSION_KEY] = user_language
```

通常，这二者结合在一起使用：先使用 django.utils.translation.activate() 为当前线程设定语言，然后修改会话，让后续请求继续使用。

如果不使用会话，设定的语言保存在一个 cookie 中，其名称由 LANGUAGE_COOKIE_NAME 配置。例如：

```
from django.utils import translation
from django import http
from django.conf import settings
user_language = 'fr'
translation.activate(user_language)
response = http.HttpResponse(...)
response.set_cookie(settings.LANGUAGE_COOKIE_NAME, user_language)
```

18.9 在视图和模板之外使用翻译

Django 为视图和模板提供了丰富的国际化工具，但是并没有限制只能用于 Django 代码。Django 的翻译机制能把任何文本翻译成 Django 支持的语言（当然，相应的翻译目录要存在）。

你可以加载一个翻译目录，激活它，把文本翻译成你选择的语言，但是别忘了切换回原语言，因为翻译目录在线程中激活，会影响同一个线程中运行的所有代码。

例如：

```
from django.utils import translation
def welcome_translated(language):
    cur_language = translation.get_language()
    try:
        translation.activate(language)
        text = translation.ugettext('welcome')
    finally:
        translation.activate(cur_language)
    return text
```

此时，不管 LANGUAGE_CODE 设置和中间件设定的语言是什么，如果指定语言为“de”，得到的是“Willkommen”。

这里涉及的几个函数是：django.utils.translation.get_language()，返回当前线程中使用的语言；django.utils.translation.activate()，在当前线程中激活指定的翻译目录；django.utils.translation.check_for_language()，检查 Django 是否支持指定的语言。

18.10 实现方式说明

18.10.1 Django 翻译机制的特殊之处

Django 的翻译机制采用 Python 标准库中的 `gettext` 模块。如果你了解 `gettext`，将会发现 Django 处理翻译的方式有些特别之处：

- 字符串域是 `django` 或 `djangojs`。域用于区分不同程序在同一个位置（通常是 `/usr/share/locale/`）存储的消息文件。`django` 域针对 Python 和模板代码的翻译字符串，加载到全局翻译目录中。`djangojs` 域只针对 JavaScript 翻译目录，目的是让 JavaScript 翻译目录尽量保持小体积。
- Django 不单独使用 `xgettext`，而是使用 Python 对 `xgettext` 和 `msgfmt` 的包装程序。这样做基本上是为了操作方便。

18.10.2 Django 如何发现语言偏好设置

准备好翻译之后，或者直接使用 Django 自带的翻译时，要激活翻译。

在这背后，Django 采用一种十分灵活的方式判断该使用哪个语言——全局或针对各个用户，抑或二者兼具。

若想全局设定语言偏好设置，使用 `LANGUAGE_CODE` 设置。Django 把这个设置设定的语言作为默认语言，即本地化中间件找不到匹配的翻译时所做的补救措施（参见下文）。

如果只想让 Django 使用你的母语，只需设定 `LANGUAGE_CODE` 设置，并确保相应的消息文件及编译后的版本（`.mo`）存在。

如果想让用户自行选择语言，还要使用 `LocaleMiddleware`。`LocaleMiddleware` 根据请求中的数据选择语言，然后为用户定制内容。

若想使用 `LocaleMiddleware`，把 `'django.middleware.locale.LocaleMiddleware'` 添加到 `MIDDLEWARE_CLASSES` 设置中。中间件是有一定顺序的，请遵循下述指导方针：

- 确保是头几个中间件之一。
- 应该放在 `SessionMiddleware` 后面，因为 `LocaleMiddleware` 要使用会话数据。而且要放在 `CommonMiddleware` 前面，因为 `CommonMiddleware` 要知道激活了哪个语言才能解析所请求的 URL。
- 如果使用 `CacheMiddleware`，把 `LocaleMiddleware` 放在它后面。

例如，`MIDDLEWARE_CLASSES` 设置可能是这样的：

```
MIDDLEWARE_CLASSES = [
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.middleware.locale.LocaleMiddleware',
    'django.middleware.common.CommonMiddleware',
]
```

中间件的更多说明参见第 17 章。

`LocaleMiddleware` 通过下述法则确定用户的语言偏好设置：

- 首先，查看所请求 URL 的语言前缀。仅当 URL 配置中使用了 `i18n_patterns` 函数才会这么做。关于语言前缀，以及如何国际化 URL 模式，参见18.6 节。
- 如果无法确定，查看当前用户会话中的 `LANGUAGE_SESSION_KEY` 键。

- 如果无法确定，查看一个 cookie。这个 cookie 的名称由 LANGUAGE_COOKIE_NAME 设置设定。（默认名称为 django_language。）
- 如果无法确定，查看 Accept-Language 首部。这个首部由浏览器发送，作用是告诉服务器可接受的语言（按优先级顺序列出）。Django 依次尝试这个首部中的语言，直到找出可用的翻译。
- 如果无法确定，使用全局的 LANGUAGE_CODE 设置。

注意：

- 在每一步中，语言偏好设置的值都应该是字符串形式的标准语言格式。例如，巴西葡萄牙语是 'pt-br'。
- 如果基语言可用，而子语言不可用，Django 将使用基语言。例如，如果用户指定的语言是 de-at（奥地利德语），而应用程序只提供了 de 的翻译，Django 将使用 de。
- 只能选择 LANGUAGES 设置中列出的语言。如果想把选择限制在几种语言之间（因为应用程序没有提供所有语言），把 LANGUAGES 设为一个语言列表。例如：

```
LANGUAGES = [
    ('de', _('German')),
    ('en', _('English')),
]
```

  这样就限制只能在德语和英语（及其子语言，如 de-ch 或 en-us）之间选择。

- 如果像前一点所说，定义了 LANGUAGES 设置，可以把语言名称标记为翻译字符串，不过要使用 ugettext_lazy()，而不能使用 ugettext()，以免循环导入。下面举个例子：

```
from django.utils.translation import ugettext_lazy as _

    LANGUAGES = [
        ('de', _('German')),
        ('en', _('English')),
    ]
```

LocaleMiddleware 确定用户的语言偏好设置之后，将其存入每个 HttpRequest 对象的 request.LANGUAGE_CODE 属性中。这样，在视图代码中就可以读取了。下面举个例子：

```
from django.http import HttpResponse

def hello_world(request, count):
    if request.LANGUAGE_CODE == 'de-at':
        return HttpResponse("You prefer to read Austrian German.")
    else:
        return HttpResponse("You prefer to read another language.")
```

注意，静态翻译（不经中间件）的语言在 settings.LANGUAGE_CODE 中，而动态翻译（经过中间件）的语言在 request.LANGUAGE_CODE 中。

18.10.3 Django 如何发现翻译

运行时，Django 把字面量翻译合并到一起，构建一个统一的翻译目录，存储在内存中。为此，Django 遵循下述规则，确定以何种顺序扫描不同的文件路径，加载编译好的翻译文件（.mo），以及以何种顺序加载相同字面量的多个翻译：

- LOCALE_PATHS 设置中列出的目录优先级最高，而且排在前面的目录优先级比排在后面的目录高。
- 然后，查看 INSTALLED_APPS 设置中列出的各个应用，使用各个应用 locale 目录中的翻译。这个设置中排在前面的应用比排在后面的优先级高。
- 最后，Django 在 django/conf/locale 目录中提供了一些基础翻译，做为一种后备机制。

包含翻译的目录应该以区域名称命名，例如 de、pt_BR、es_AR，等等。

这样，你便可以编写自带翻译的应用程序，而且可以在项目中覆盖基础翻译。此外，也可以构建一个由多个应用构成的大型项目，把所有翻译都放在一个通用的消息文件中。选择权在你手中。

消息文件目录的结构类似：

- 在 LOCALE_PATHS 设置列出的各个路径中搜索 <language>/LC_MESSAGES/django.(po|mo)。
- $APPPATH/locale/<language>/LC_MESSAGES/django.(po|mo)。
- $PYTHONPATH/django/conf/locale/<language>/LC_MESSAGES/django.(po|mo)。

消息文件使用 django-admin makemessages 命令创建。gettext 使用的二进制 .mo 文件使用 django-admin compilemessages 命令生成。

运行 django-admin compilemessages 命令时还可以让编译器处理 LOCALE_PATHS 设置中列出的所有目录。

18.11 接下来

下一章讨论 Django 中的安全措施。

第 19 章 安全保护

专业的 Web 应用程序开发者的一项重要职责是确保网站的安全性。

如今，Django 框架已经十分成熟，大多数常见的安全问题都由框架本身以某种方式解决了，然而，安全措施难保万全，因为随时都有新的威胁出现，所以，作为 Web 开发者的我们要以身作则，确保网站和 Web 应用的安全性。

Web 安全性是个涉及面很广的话题，很难在一章的内容中深入探讨。本章简述 Django 提供的安全特性，并给出一些安全方面的建议，确保 Django 驱动的网站在 99% 的情况下是安全的，Web 安全的最新情况则有你自己去跟踪。

如果想详细了解 Web 安全，可以看看 Django 的安全问题存档，以及维基百科中的“Web application security”词条。

19.1 Django 内置的安全特性

19.1.1 跨站脚本防护

跨站脚本攻击（Cross Site Scripting，XSS）的原理是一个用户在另一个用户的浏览器中注入客户端脚本。

恶意脚本通常存储在数据库中，当用户查看页面或点击链接时再读取出来，在用户的浏览器中执行。然而，只要没有对页面中的数据做好净化，任何不可信的数据源，如 cookie 或 Web 服务，都有可能成为 XSS 攻击的源头。

使用 Django 模板能防护大多数 XSS 攻击。然而，一定要了解它提供的保护措施，以及不足之处。

Django 模板会过滤字符，尤其是对 HTML 而言危险的字符。这能防止用户输入多数恶意内容，但却不是万无一失的。例如，无法防止输入下述内容：

```
<style class={{ var }}>...</style>
```

如果 `var` 的值是 `'class1 onmouseover=javascript:func()'`，那么这段输入就可能导致执行非法的 JavaScript 代码（取决于浏览器如何渲染不完美的 HTML）。（把属性的值放在引号中能解决这个问题。）

此外，自定义的模板标签、`safe` 标签、`mark_safe` 和禁用 `autoescape` 时一定要慎重使用 `is_safe`。

如果使用模板系统输出 HTML 之外的格式，要注意，此时需要转义的字符和单词可能完全不同。

在数据库中存储 HTML 时也要十分小心，尤其是要取出并显示 HTML 时。

19.1.2 跨站请求伪造防护

跨站请求伪造（Cross Site Request Forgery，CSRF）攻击的原理是在用户不知情或未准许的情况下以用户的身份执行操作。

Django 内置的措施能防护多种 CSRF 攻击，前提是你要适时启用。然而，与任何缓和技术一样，这些措施也有不足之处。

例如，CSRF 模块可以全局或在具体的视图上禁用。如果想禁用，一定要心中有数。如果网站的某些二级域名不在你的控制之内，还有其他局限。

CSRF 防护的原理是，检查每个 `POST` 请求中的一次性令牌。这样能确保恶意用户无法重放提交的表单，因此也就无法在用户不知情的情况下提交表单。除非用户获知了一次性令牌，而那是每个用户专用的（存储在 cookie 中）。

使用 HTTPS 时，`CsrfViewMiddleware` 会检查 HTTP `Referer` 首部中的 URL 是否来自相同的源（包括二级域名和端口）。HTTPS 增强了安全性，因此只要可能就应该使用 HTTPS，而且要把不安全的连接转发到安全的连接，并为支持的浏览器启用 HSTS。

除非真的有必要，不要轻易使用 `csrf_exempt` 装饰器装饰视图。

Django 的 CSRF 中间件和模板标签为防护跨站请求伪造提供了易于使用的措施。

CSRF 攻击的第一道防线是确保 `GET` 请求（以及其他"安全的"请求方法，参见 HTTP 1.1 规范 RFC 2616 中的"9.1.1 Safe Methods"）没有副作用。"不安全的"请求方法，如 `POST`、`PUT` 和 `DELETE` 可以参照下述步骤保护。

用法

在视图中防护 CSRF 的步骤如下：

1. `MIDDLEWARE_CLASSES` 设置默认已激活 CSRF 中间件。如果覆盖了那个设置，记得要把 `'django.middleware.csrf.CsrfViewMiddleware'` 放在所有假定已经处理过 CSRF 攻击的视图中间件前面。如果禁用了这个中间件（不推荐），可以在想保护的视图上使用 `csrf_protect()` 装饰器（参见下文）。
2. 在通过 `POST` 方法发送的表单中，如果表单的目标地址是内部 URL，在 `<form>` 元素中使用 `csrf_token` 标签。例如：

   ```
   <form action="." method="post">
       {% csrf_token %}
   ```

   如果 `POST` 表单的模板地址是外部 URL，不能这么做；否则，CSRF 令牌就泄露了，导致网站存在漏洞。
3. 确保在对应的视图函数中使用 `'django.template.context_processors.csrf'` 上下文处理器。通常，方法有如下两种：

   a. 使用 `RequestContext`，它始终使用 `'django.template.context_processors.csrf'`（不管在 `TEMPLATES` 设置中配置的是哪个模板上下文处理器）。如果使用的是泛视图或扩展应用，无需自己动手，因为它们使用的全是 `RequestContext`。

   b. 自己动手导入上下文处理器，使用它生成 CSRF 令牌，再将其添加到模板上下文中。例如：

   ```
   from django.shortcuts import render_to_response
    from django.template.context_processors import csrf

    def my_view(request):
        c = {}
        c.update(csrf(request))
        # ... 其他视图代码
        return render_to_response("a_template.html", c)
   ```

   你也可以自己编写 `render_to_response()`，实现这一步。

Ajax

虽然上述方法可以在 Ajax POST 请求中使用，但是有些不便：每个 POST 请求都不能忘了在 POST 数据中发送 CSRF 令牌。鉴于此，Django 提供了另外一种方法：为 `XMLHttpRequest` 设定自定义的 `X-CSRFToken` 首部，将其值设为 CSRF 令牌。通常，这样更简单，因为很多 JavaScript 框架都提供了钩子，能轻易为每个请求设定首部。

为此，首先要获取 CSRF 令牌。建议从 `csrftoken` cookie 中获取。按上述步骤为视图启用 CSRF 防护后，会设定这个 cookie。

存储 CSRF 令牌的 cookie 默认名为 `csrftoken`，不过也可以使用 `CSRF_COOKIE_NAME` 设置自定义。

获取令牌的方法很简单：

```
// 使用 jQuery
function getCookie(name) {
    var cookieValue = null;
    if (document.cookie && document.cookie != '') {
        var cookies = document.cookie.split(';');
        for (var i = 0; i < cookies.length; i++) {
            var cookie = jQuery.trim(cookies[i]);
            // cookie 的名称以指定字符串开头吗?
            if (cookie.substring(0, name.length + 1) == (name + '=')) {
                cookieValue =   decodeURIComponent(cookie.substring(name.length + 1));
                break;
            }
        }
    }
    return cookieValue;
}
var csrftoken = getCookie('csrftoken');
```

上述代码可以通过 jQuery Cookie 插件简化，无需定义 `getCookie` 函数：

```
var csrftoken = $.cookie('csrftoken');
```

提示

DOM 中也有 CSRF 令牌，但是仅当在模板中使用 `csrf_token` 时才有。cookie 中存储的令牌才是正源，`CsrfViewMiddleware` 首选 cookie 中的令牌，而非 DOM 中的。不管这么多，只要 DOM 中有令牌，cookie 中就有，所以应该使用 cookie 中的令牌。

提醒

如果视图渲染的模板中没有 `csrf_token` 标签，Django 可能不会设定 CSRF 令牌 cookie。动态添加到页面中的表单往往是这种情况。为了解决这个问题，Django 提供了一个视图装饰器，强制设定那个 cookie：`ensure_csrf_cookie()`。

最后，在 Ajax 请求中设定 `X-CSRFToken` 首部。为了防止把 CSRF 令牌发给其他域名，在 jQuery 1.5.1 及以上版本中应该使用 `settings.crossDomain`。

```
function csrfSafeMethod(method) {
```

```
    // 这些 HTTP 方法不需要 CSRF 防护
    return (/^(GET|HEAD|OPTIONS|TRACE)$/.test(method));
}
$.ajaxSetup({
    beforeSend: function(xhr, settings) {
        if (!csrfSafeMethod(settings.type) && !this.crossDomain) {
            xhr.setRequestHeader("X-CSRFToken", csrftoken);
        }
    }
});
```

其他模板引擎

如果使用的模板引擎不是 Django 内置的，可以自己动手在表单中设定 CSRF 令牌。当然，首先要保证模板上下文中有令牌。

例如，使用 Jinja2 模板语言时，可以这样编写表单：

```
<div style="display:none">
    <input type="hidden" name="csrfmiddlewaretoken" value="{{ csrf_token }}">
</div>
```

也可以像上述 Ajax 代码那样使用 JavaScript 获取 CSRF 令牌的值。

装饰器方法

如果不想使用 `CsrfViewMiddleware` 提供的全面防护措施，可以在需要保护的视图上使用 `csrf_protect` 装饰器，其作用与 `CsrfViewMiddleware` 完全一样。输出中有 CSRF 令牌的视图，以及接受 `POST` 表单数据的视图（二者通常是同一个视图，但也有例外）都要使用这个装饰器。

> **提醒**
>
> 不建议只使用这个装饰器，因为一旦忘记就暴露了安全漏洞。以防万一，可以二者同时使用，由此增加的消耗很少。

用法：

```
from django.views.decorators.csrf import csrf_protect
from django.shortcuts import render

@csrf_protect
def my_view(request):
    c = {}
    # ...
    return render(request, "a_template.html", c)
```

如果使用基于类的视图，请参阅 Django 文档。

请求被拒

默认情况下，如果入站请求未通过 `CsrfViewMiddleware` 的检查，将返回“403 Forbidden”响应。出现这种情况，通常说明真的遭到跨站请求伪造攻击了，或者是程序设计有问题，`POST` 表单中没有 CSRF 令牌。

然而，错误页面不是十分友好，因此你可能想自己编写视图处理这种情况。为此，只需设定 CSRF_FAILURE_VIEW 设置。

原理

CSRF 防护的背后是这样的：

- CSRF cookie 是一个随机值（与会话无关的一次性值），其他网站无法访问。这个 cookie 由 CsrfViewMiddleware 设定。这个 cookie 的作用是永久的，但是因为无法设定永不过期的 cookie，所以每个响应都会调用 django.middleware.csrf.get_token()（内部使用这个函数获取 CSRF 令牌），随响应一起发送。
- 每个外送的 POST 表单都有一个名为“csrfmiddlewaretoken”的隐藏字段。这个字段的值就是 CSRF cookie 中的值。这个字段由一个模板标签添加。
- 入站请求，只要使用的不是 HTTP GET、HEAD、OPTIONS 或 TRACE，必须有 CSRF cookie，而且必须有值正确的“csrfmiddlewaretoken”字段。否则，用户会得到 403 错误。这项检查由 CsrfViewMiddleware 实施。
- 此外，CsrfViewMiddleware 会对 HTTPS 请求做严格的来源检查。这是为了避免在 HTTPS 中使用与会话无关的一次性值时导致的中间人攻击，因为通过 HTTPS 与网站通信的客户端接受 HTTP Set-Cookie 首部（真遗憾）。（HTTP 请求不做来源检查，因为在 HTTP 中 Referer 首部不太可靠。）这样能确保只有源自自己网站的表单才能把数据发送回来。
- 故意忽略 GET 请求（以及 RFC 2616 定义的其他“安全”请求）。这些请求决不能带有潜在有害的副作用，因此即便遭受 CSRF 攻击，GET 请求也没什么危害。RFC 2616 把 POST、PUT 和 DELETE 定义为“不安全的”请求方法，除此之外的其他所有方法都假定是安全的；不安全的方法要尽力防护。

缓存

如果在模板中使用了 csrf_token 标签（或者以其他方式调用了 get_token 函数），CsrfViewMiddleware 会添加一个 cookie，并为响应添加 Vary: Cookie 首部。这意味着，如果使用得当（UpdateCacheMiddleware 放在其他所有中间件前面），CsrfViewMiddleware 能与缓存中间件协同工作。

然而，如果在具体的视图上使用缓存装饰器，那么 CSRF 中间件就来不及设定 Vary 首部或 CSRF cookie，从而缓存的响应中二者都没有。

对需要插入 CSRF 令牌的视图来说，应该先使用 django.views.decorators.csrf.csrf_protect() 装饰器：

```
from django.views.decorators.cache import cache_page
from django.views.decorators.csrf import csrf_protect

@cache_page(60 * 15)
@csrf_protect
def my_view(request):
    ...
```

如果使用基于类的视图，请参阅 Django 文档。

测试

CsrfViewMiddleware 往往会给测试视图函数带来麻烦，因为每个 POST 请求都要带有 CSRF 令牌。鉴于此，Django 的 HTTP 测试客户端做了特殊处理，设定了一个旗标，让这个中间件和 csrf_protect 装饰器放松警惕，不再拒绝请求。除此之外，HTTP 测试客户端的行为都与前文所述的一样（例如，会发送 CSRF cookie）。

如果某些情况下想让测试客户端检查 CSRF，可以创建一个测试客户端实例，要求它检查 CSRF：

```
>>> from django.test import Client
>>> csrf_client = Client(enforce_csrf_checks=True)
```

不足之处

网站的二级域名能在客户端为整个域名设定 cookie。有了 cookie 和其中的令牌，二级域名就能绕过 CSRF 防线。唯一能避免这个问题的措施是，确保二级域名在受信任的人手中（或者至少不能设定 cookie）。

注意，即便不考虑 CSRF，也不能把二级域名交给不可信的一方，因为还有其他漏洞（如会话固定）在现今的浏览器中无法轻易补上。

边缘情况

某些视图有不同寻常的需求，因此不能使用这里所述的常规方式处理。遇到非常规情况时，可以使用几个实用工具。这些工具的使用场景在下一节中说明。

19.1.3 CSRF 实用工具

下述各示例假定你使用的是基于函数的视图。如果使用基于类的视图，请参阅 Django 文档。

`django.views.decorators.csrf.csrf_exempt(view)`

多数视图需要 CSRF 防护，但少数不需要。不要禁用 CSRF 中间件，然后在需要防护的视图上使用 `csrf_protect` 装饰器；正确的做法是启用 CSRF 中间件，使用 `csrf_exempt()` 装饰器。

这个装饰器指明视图免受 CSRF 中间件的保护。例如：

```
from django.views.decorators.csrf import csrf_exempt
from django.http import HttpResponse

@csrf_exempt
def my_view(request):
    return HttpResponse('Hello world')
```

`django.views.decorators.csrf.requires_csrf_token(view)`

有些情况下，`CsrfViewMiddleware.process_view` 可能不在视图之前运行（例如 404 和 500 处理程序），但是仍想在表单中使用 CSRF 令牌。

通常，如果 `CsrfViewMiddleware.process_view` 或等效的函数（如 `csrf_protect`）没有运行，`csrf_token` 模板标签将不起作用。视图装饰器 `requires_csrf_token` 可以确保这个标签一定起作用。这个装饰器的原理与 `csrf_protect` 类似，但是从不拒绝入站请求。

例如：

```
from django.views.decorators.csrf import requires_csrf_token
from django.shortcuts import render

@requires_csrf_token
def my_view(request):
    c = {}
```

```
    # ...
    return render(request, "a_template.html", c)
```

还有些视图不受保护，被 csrf_exempt 豁免了，但是仍需引入 CSRF 令牌。此时，在 csrf_exempt() 后面使用 requires_csrf_token()（即 requires_csrf_token 放在最内层）。

最后，有些视图仅在满足特定条件时需要 CSRF 防护，而其他情况下则不需要。这种需求的一种解决方法是，在整个视图函数上使用 csrf_exempt()，然后在需要 CSRF 防护的代码路径上使用 csrf_protect()。

例如：

```
from django.views.decorators.csrf import csrf_exempt, csrf_protect

@csrf_exempt
def my_view(request):

    @csrf_protect
    def protected_path(request):
        do_something()

    if some_condition():
       return protected_path(request)
    else:
       do_something_else()
```

django.views.decorators.csrf.ensure_csrf_cookie(view)

这个装饰器强制视图发送 CSRF cookie。比如说，如果通过 Ajax 发起 POST 请求的页面中没有包含 csrf_token 的 HTML 表单，此时就需要发送 CSRF cookie。这种需求的解决方法是在视图上使用 ensure_csrf_cookie() 装饰器。

扩展应用和可复用的应用

因为开发者能够禁用 CsrfViewMiddleware，所以扩展应用中的全部相关视图都使用 csrf_protect 装饰器确保应用能防护 CSRF 攻击。其他可复用的应用开发者，如果想得到这种保障，建议也在视图上使用 csrf_protect 装饰器。

CSRF 设置

Django 的 CSRF 防护行为受几个设置的控制：

- CSRF_COOKIE_AGE
- CSRF_COOKIE_DOMAIN
- CSRF_COOKIE_HTTPONLY
- CSRF_COOKIE_NAME
- CSRF_COOKIE_PATH
- CSRF_COOKIE_SECURE
- CSRF_FAILURE_VIEW

各个设置的详细说明参见附录 D。

19.1.4 SQL 注入防护

SQL *注入*（injection）是一种攻击类型，原理是恶意用户在数据库中执行任意的 SQL 代码。SQL 注入可能导致记录被删，或者导致数据泄露。

使用 Django 的查询集合，底层的数据库驱动会正确转义得到的 SQL。然而，Django 也允许开发者编写原始查询或执行自定义的 SQL。使用这两个功能时要谨慎一些，应该始终转义由用户控制的参数。此外，使用 `extra()` 时要格外小心。

19.1.5 点击劫持防护

点击劫持（clickjacking）是一种攻击类型，原理是在框架（frame）中内嵌恶意网站。攻击者把恶意网站内嵌在隐蔽的框架中，骗取用户的任性，让他们点击，这就是点击劫持。

Django 通过 `XFrameOptionsMiddleware` 防护点击劫持，在支持 `X-Frame-Options` 首部的浏览器中，无法把网站内嵌在框架中。这项防护措施可以在具体的视图上禁用，还可以配置发送的首部值。

如果网站无需通过框架内嵌到第三方网站中，或者只允许网站的一小部分内嵌到第三方网站中，强烈建议启用这个中间件。

点击劫持示例

假设有个在线商店，在一个页面上已登录的用户可以点击"Buy Now"，购买商品。为了方便操作，用户选择记住登录状态。攻击者可以在自己的网站中创建一个"I Like Ponies"按钮，然后在透明的框架中加载在线商店的页面，把不可见的"Buy Now"按钮叠放到"I Like Ponies"按钮上。用户访问攻击者的网站，点击"I Like Ponies"，此时便会意外点击"Buy Now"按钮，在不知情的情况下购买了商品。

避免点击劫持

现代的浏览器会遵照 HTTP `X-Frame-Options` 首部的设定，判断是否允许在框架中加载资源。如果响应中的这个首部值为 `SAMEORIGIN`，那么浏览器只允许来自同一网站的请求在框架中加载资源。如果首部的值为 `DENY`，浏览器将禁止在框架中加载资源，不管请求来自哪个网站都不允许。

为了在响应中添加这个首部，Django 提供了两种简单的方式：

- 一个简单的中间件，为所有响应设定这个首部。
- 一系列视图装饰器，用于覆盖中间件的设置，或者只在特定的视图上设定这个首部。

用法

为所有响应设定 `X-Frame-Options`

若想为网站中的所有响应设定 `X-Frame-Options` 首部，在 `MIDDLEWARE_CLASSES` 设置中添加 `'django.middleware.clickjacking.XFrameOptionsMiddleware'`：

```
MIDDLEWARE_CLASSES = [
    # ...
    'django.middleware.clickjacking.XFrameOptionsMiddleware',
    # ...
]
```

`startproject` 命令生成的设置文件默认启用了这个中间件。

默认情况下，这个中间件把每个出站响应的 `X-Frame-Options` 首部设为 `SAMEORIGIN`。如果想设为 `DENY`，像这样设定 `X_FRAME_OPTIONS` 设置：

```
X_FRAME_OPTIONS = 'DENY'
```

启用这个中间件后，可能有些视图不需要设定 `X-Frame-Options` 首部。此时，可以使用一个视图装饰器告知中间件，别在视图上设定 `X-Frame-Options` 首部：

```
from django.http import HttpResponse
from django.views.decorators.clickjacking import xframe_options_exempt

@xframe_options_exempt
def ok_to_load_in_a_frame(request):
    return HttpResponse("This page is safe to load in a frame on any site.")
```

在具体的视图上设定 `X-Frame-Options`

为了便于在具体的视图上设定 `X-Frame-Options` 首部，Django 提供了下述几个装饰器：

```
from django.http import HttpResponse
from django.views.decorators.clickjacking import xframe_options_deny
from django.views.decorators.clickjacking import xframe_options_sameorigin

@xframe_options_deny
def view_one(request):
    return HttpResponse("I won't display in any frame!")

@xframe_options_sameorigin
def view_two(request):
    return HttpResponse("Display in a frame if it's from the same
      origin as me.")
```

注意，这些装饰器可以与 `XFrameOptionsMiddleware` 结合起来使用，通过装饰器覆盖中间件的全局行为。

不足之处

只有在现代的浏览器中 `X-Frame-Options` 首部才能防护点击劫持。旧浏览器直接忽略这个首部，需要使用其他方式防护点击劫持。

支持 `X-Frame-Options` 的浏览器

- Internet Explorer 8+
- Firefox 3.6.9+
- Opera 10.5+
- Safari 4+
- Chrome 4.1+

19.1.6 SSL/HTTPS

把网站部署在 HTTPS 后面对安全性肯定更好，但并不适用所有情况。如果不使用 HTTPS，恶意用户可以嗅探身份认证凭据等在客户端和服务器之间传输的信息，有时攻击者甚至可以篡改发给二者的数据。

如果想使用 HTTPS 提供的保护措施，而且在服务器上已经启用 HTTPS，可能还需要做以下几件事：

- 如有必要，设定 `SECURE_PROXY_SSL_HEADER`，确保全面理解涉及的提醒。如果不设定这个设置，可能暴漏 CSRF 漏洞；未正确设置也有一定的危险。
- 把 HTTP 请求重定向到 HTTPS。这一步可以通过一个自定义中间件实现。注意，`SECURE_PROXY_SSL_HEADER` 有个坑。如果使用反向代理，让主 Web 服务器重定向更容易，也更安全。
- 使用"安全的"cookie。如果浏览器一开始是通过 HTTP 连接的（多数浏览器默认如此），现有的 cookie 可能遭到泄露。鉴于此，应该把 `SESSION_COOKIE_SECURE` 和 `CSRF_COOKIE_SECURE` 设置设为 `True`。这么做的目的是让浏览器只通过 HTTPS 发送那两个 cookie。注意，这样设定之后，在 HTTP 连接中会话不可用，而且 CSRF 防护措施拒绝接受通过 HTTP 发送的 `POST` 数据（如果把 HTTP 流量重定向到 HTTPS，那就没问题）。
- 使用 HTTP Strict Transport Security（HSTS）。HSTS 是一个 HTTP 首部，告知浏览器，针对指定网站的后续请求都使用 HTTPS（参见下文）。我们已经把 HTTP 请求重定向到 HTTPS 了，再加上 HSTS，只要成功连接了一次，后续请求就能享用 SSL 增添的安全措施。HSTS 通常在 Web 服务器上配置。

HTTP Strict Transport Security

如果你的网站只允许通过 HTTPS 访问，可以设定 `Strict-Transport-Security` 首部，告诉现代的浏览器，（在一段时间内）拒绝通过不安全的连接访问你的域名。这样能降低被某些 SSL 层面的中间人攻击的风险。

如果把 `SECURE_HSTS_SECONDS` 设置设为非零整数值，`SecurityMiddleware` 会为所有 HTTPS 响应设定这个首部。

启用 HSTS 时，最好先使用小一点的值测试，例如设为一小时：`SECURE_HSTS_SECONDS = 3600`。只要 Web 浏览器发现有 HSTS 首部，就会在一段时间内拒绝通过不安全的连接（HTTP）访问你的域名。

确保网站中的所有静态资源都能正确伺服之后（即 HSTS 没有造成影响），最好增加时长（经常设为 31536000 秒，即一年），让不常访问的访客也得到保护。

此外，如果把 `SECURE_HSTS_INCLUDE_SUBDOMAINS` 设置设为 `True`，`SecurityMiddleware` 会在 `Strict-Transport-Security` 首部中添加 `includeSubDomains` 标签。这是推荐的做法（所有二级域名也只能通过 HTTPS 访问），如若不然，通过不安全的连接访问二级域名还是可能存在漏洞。

> **提醒**
>
> HSTS 策略应用于整个域名，而不只是设定 `Strict-Transport-Security` 首部的那个 URL。因此，仅当整个域名都通过 HTTPS 伺服时才应该使用 HSTS。严格遵守 HSTS 首部的浏览器遇到过期的、自签名的或无效的 SSL 证书时拒绝用户跳过提醒，不允许继续访问。使用 HSTS 时，要确保证书始终有效！

> **提示**
>
> 如果网站部署在负载均衡程序或反向代理服务器后面，而响应中没有 `Strict-Transport-Security` 首部，可能是因为 Django 没将其视作安全的连接。此时可能要设定 `SECURE_PROXY_SSL_HEADER` 设置。

19.1.7 验证 Host 首部

某些情况下，Django 使用客户端提供的 Host 首部构建 URL。虽然为了防止跨站脚本攻击，这个首部的值做了净化，但是在跨站请求伪造、缓存中毒攻击和邮件链接中毒中，Host 首部的值可能是虚假的。

尽管 Web 服务器的配置看似是安全的，但是依然会受到虚假 Host 首部的影响。所以，Django 会验证 Host 首部，调用 django.http.HttpRequest.get_host() 方法时，检查得到的值是否在 ALLOWED_HOSTS 设置中。

只有调用 get_host() 方法时才会执行这个检查。如果直接从 request.META 中读取 Host 首部，就跳过了这项安全保护措施。

19.1.8 会话安全性

CSRF 防护措施的不足之处是，要求二级域名不在不可信的用户手中。类似地，django.contrib.sessions 也有这样的缺点。详情参见文档中对会话安全性的说明。

19.1.9 用户上传的内容

- 如果你的网站允许上传文件，强烈建议在 Web 服务器的配置中把上传文件的大小限制为合理的值，以防遭到拒绝服务（Denial of Service，DoS）攻击。在 Apache 中可以使用 LimitRequestBody 指令设定。
- 如果自己伺服静态文件，一定要禁用能把静态文件当做代码执行的处理程序，如 Apache 的 mod_php。难道你想让用户随意执行精心制作的文件中的代码?
- 如果不遵守安全最佳实践，Django 对上传媒体的处理有些漏洞。比如说，如果 HTML 文件包含有效的 PNG 首部，随后是恶意 HTML 代码，那么上传时会把它视作图像。对 Django 用来处理 ImageField 的图像库（Pillow）来说，这个文件能通过验证。渲染时，这个文件可能会作为 HTML 显示，这取决于 Web 服务器的类型和配置。框架层虽然没有万全之策，无法验证所有用户上传的文件包含什么内容，但是我们可以做几件事，降低被攻击的风险：

  1. 使用单独的顶级域名或二级域名伺服用户上传的文件能避免其中一类攻击，例如同源策略阻拦的攻击（如跨站脚本攻击）。假如你的网站部署在 example.com 域名上，可以在 usercontent-example.com 上伺服上传的文件（通过 MEDIA_URL 设置设定），而不一定非得在二级域名（如 usercontent.example.com）上伺服。
  2. 此外，应用程序可以定义一个白名单，指明允许用户上传的文件扩展名，然后再配置 Web 服务器，只伺服那些文件。

19.2 其他安全建议

- 虽然 Django 自带了稳固的安全保护措施，但是依然要采用正确的方式部署应用程序，利用 Web 服务器、操作系统和其他组件提供的安全保护措施。
- 记得把 Python 代码放在 Web 服务器的文档根目录之外。这样能避免不小心以纯文本伺服（或执行）Python 代码。
- 谨慎处理用户上传的文件。
- Django 不限制验证用户身份的次数。为了防止暴力攻击身份验证系统，可以考虑通过 Django 插件或 Web 服务器模块限制这种请求的次数。
- 保护好 SECRET_KEY 设置。
- 最好使用防火墙限制对缓存系统和数据库的访问。

19.2.1 安全问题存档

Django 的开发团队坚定承诺，他们会负责报告揭露安全相关的问题。这一点在 Django 的安全策略中有体现。

作为这个承诺的一部分，他们维护着已经修正和揭露出来的问题列表。最新的列表参见 Django 文档。

19.2.2 加密签名

Web 应用程序安全性的黄金法则是，绝不信任来自不可信源的数据。有时，可能需要通过不可信的媒介传递数据。此时，可以加密签名要传递的数据，因为只要篡改就能发现。

根据 Web 应用程序的常见需求，Django 提供了用于签名的低层 API，以及用于设定和读取签名 cookie 的高层 API。

下述情况也能用到签名：

- 生成"恢复我的账号" URL，发给忘记密码的用户。
- 确保隐藏的表单字段中的数据没有被篡改。
- 生成一次性保密 URL，用于临时访问受保护的资源。例如，下载用户付费购买的文件。

保护 SECRET_KEY

使用 `startproject` 命令新建 Django 项目时，自动生成的 `settings.py` 文件把 `SECRET_KEY` 设为一个随机值。这个值是保护签名数据的关键，一定要保护好，如果被攻击者获取，就能用它生成签名数据。

使用低层 API

Django 提供的签名方法在 `django.core.signing` 模块中。若想签名一个值，先实例化一个 `Signer` 实例：

```
>>> from django.core.signing import Signer
>>> signer = Signer()
>>> value = signer.sign('My string')
>>> value
'My string:GdMGD6HNQ_qdgxYP8yBZAdAIV1w'
```

签名附加到字符串的末尾，通过冒号与原字符串分开。可以使用 `unsign` 方法获取原值：

```
>>> original = signer.unsign(value)
>>> original
'My string'
```

一旦签名或原值以某种方式修改了，抛出 `django.core.signing.BadSignature` 异常：

```
>>> from django.core import signing
>>> value += 'm'
>>> try:
...     original = signer.unsign(value)
... except signing.BadSignature:
...     print("Tampering detected!")
```

默认情况下，`Signer` 类使用 `SECRET_KEY` 设置生成签名。如果想使用其他密令，把它传给 `Signer` 构造方法：

```
>>> signer = Signer('my-other-secret')
>>> value = signer.sign('My string')
>>> value
'My string:EkfQJafvGyiofrdGnuthdxImIJw'
```

django.core.signing.Signer 返回一个签名实例，使用 key 生成签名，使用 sep 分隔原值和签名。sep 的值不能在对 URL 安全的 base64 字母表（包括字母、数字、连字符和下划线）中。

使用 salt 参数

如果不希望相同的字符串每次签名都得到相同的结果，可以使用 Signer 构造方法的可选参数 salt。此时，salt 和 SECRET_KEY 都会传给计算签名的函数。

```
>>> signer = Signer()
>>> signer.sign('My string')
'My string:GdMGD6HNQ_qdgxYP8yBZAdAIV1w'
>>> signer = Signer(salt='extra')
>>> signer.sign('My string')
'My string:Ee7vGi-ING6n02gkcJ-QLHg6vFw'
>>> signer.unsign('My string:Ee7vGi-ING6n02gkcJ-QLHg6vFw')
'My string'
```

加盐后，不同的签名放在不同的命名空间中。一个命名空间（使用一个盐值）中的签名不能用于验证使用不同盐值的另一个命名空间中的纯文本。这样能避免攻击者在盐值不同的地方使用同一个签名。

与 SECRET_KEY 不同，盐值无需保密。

验证加了时间戳的值

TimestampSigner 是 Signer 的子类，在值后面附加一个签名的时间戳。这样能确认签名的值是不是在特定的时间段内创建的。

```
>>> from datetime import timedelta
>>> from django.core.signing import TimestampSigner
>>> signer = TimestampSigner()
>>> value = signer.sign('hello')
>>> value
'hello:1NMg5H:oPVuCqlJWmChm1rA2lyTUtelC-c'
>>> signer.unsign(value)
'hello'
>>> signer.unsign(value, max_age=10)
...
SignatureExpired: Signature age 15.5289158821 > 10 seconds
>>> signer.unsign(value, max_age=20)
'hello'
>>> signer.unsign(value, max_age=timedelta(seconds=20))
'hello'
```

sign(value) 对 value 签名，并附加当前时间戳。unsign(value, max_age=None) 检查 value 是否在 max_age 秒之前签名；如果不是，抛出 SignatureExpired 异常。max_age 参数的值可以是一个整数或一个 datetime.timedelta 对象。

保护复杂的数据结构

如果想保护列表、元组或字典，可以使用签名模块中的 dumps 和 loads 函数。这两个函数模拟 Python 的 pickle 模块，不过背后使用的序列化方式是 JSON。JSON 利用 pickle 格式，即便 SECRET_KEY 被盗，攻击者也无法随意执行命令。

```
>>> from django.core import signing
>>> value = signing.dumps({"foo": "bar"})
>>> value
'eyJmb28iOiJiYXIifQ:1NMg1b:zGcDE4-TCkaeGzLeW9UQwZesciI'
>>> signing.loads(value)
{'foo': 'bar'}
```

如果传入一个元组，使用 signing.loads(object) 能得到一个列表，这是 JSON 的天然特性（不区分列表和元组）：

```
>>> from django.core import signing
>>> value = signing.dumps(('a','b','c'))
>>> signing.loads(value)
['a', 'b', 'c']
```

19.2.3 安全中间件

提示

如果部署环境允许，通常最好让前端 Web 服务器负责执行 SecurityMiddleware 提供的功能。这样，不由 Django 伺服的请求（例如静态媒体文件或用户上传的文件）与由 Django 伺服的请求具有相同的保护措施。

django.middleware.security.SecurityMiddleware 为请求-响应循环增加了几项安全措施，可以通过下述设置单独启用或禁用：

- SECURE_BROWSER_XSS_FILTER
- SECURE_CONTENT_TYPE_NOSNIFF
- SECURE_HSTS_INCLUDE_SUBDOMAINS
- SECURE_HSTS_SECONDS
- SECURE_REDIRECT_EXEMPT
- SECURE_SSL_HOST
- SECURE_SSL_REDIRECT

关于安全方面的首部，以及这些设置的说明，参见第 17 章。

19.3 接下来

下一章扩充第 1 章中的快速安装指南，探讨 Django 的一些其他安装和配置方式。

第 20 章 安装 Django 的其他方式

本章介绍一些安装和维护 Django 的其他方式。首先，说明使用 SQLite 之外的数据库时如何配置，然后介绍如何升级 Django，以及自己动手安装 Django 的方式。最后，介绍如何安装 Django 的开发版本，以防你想试验 Django 最新开发的功能。

20.1 使用其他数据库

如果想使用 Django 的数据库 API 所提供的功能，要确保运行着数据库服务器。Django 支持很多不同的数据库服务器，官方提供支持的有 PostgreSQL、MySQL、Oracle 和 SQLite。

第 21 章将详细说明在 Django 中如何连接这些数据库，但是本书不会说明如何安装各个数据库。安装方法参见各数据库网站中的文档。

如果你开发的是小型项目，或者不打算部署到生产环境，SQLite 通常是最佳选择，因为它无需运行单独的服务器。然而，SQLite 与其他数据库有很多差异，因此如果你在开发重要的项目，建议你使用计划在生产环境使用的数据库。

除了数据库后端之外，还要安装相应的 Python 数据库绑定。

- 如果使用 PostgreSQL，要安装 postgresql_psycopg2 包。详细技术细节参见 21.2 节。Windows 用户可以使用非官方的编译版本。
- 如果使用 MySQL，要安装 MySQL-python 包（1.2.1p2 或以上版本）。详情参见 21.3 节。
- 如果使用 SQLite，请阅读 21.4 节。
- 如果使用 Oracle，要安装 cx_Oracle。所支持的 Oracle 和 `cx_Oracle` 版本，以及其他关于 Oracle 后端的信息，参见 21.5 节。
- 如果使用非官方的第三方后端，请参阅相应的文档，了解更多信息。

如果想使用 Django 的 `manage.py migrate` 命令自动为模型创建数据库表（首次安装 Django 和创建项目后），要确保 Django 有权限创建或修改数据库中的表；如果计划自己动手创建数据库表，只需把 `SELECT`、`INSERT`、`UPDATE` 和 `DELETE` 权限赋予 Django。使用这些权限创建好数据库用户之后，要在项目的设置文件中设定详细信息，详情参见 `DATABASES` 设置的文档。

如果想使用 Django 的测试框架测试数据库查询，要赋予 Django 创建测试数据库的权限。

20.2 手动安装 Django

1. 从 Django 项目网站的下载页面下载最新发布版。
2. 解压下载的文件（例如 `tar xzvf Django-X.Y.tar.gz`，其中 `X.Y` 是最新发布版的版本号）。如果使用的是 Windows，可以使用命令行工具 `bsdtar`，或者使用 GUI 工具（如 7-zip）解压。
3. 进入第 2 步得到的目录（例如 `cd Django-X.Y`）。
4. 如果使用的是 Linux、macOS 或其他 Unix 变种，在 shell 中输入命令 `sudo python setup.py install`。如果使用的是 Windows，以管理员身份启动命令行，然后运行 `python setup.py install` 命令。这些命

令把 Django 安装到 Python 的 `site-packages` 目录中。

> **删除旧版本**
>
> 如果采用这种安装方式，一定要先删除以前安装的 Django（方法参见后文）。否则，安装可能会出错，例如被 Django 删除的旧版文件仍然存在。

20.3 升级 Django

20.3.1 删除旧版 Django

升级 Django 之前要先把旧版本删除，然后再安装新版本。

如果 Django 之前是使用 `pip` 或 `easy_install` 安装的，再使用它们安装新版就无需自己动手删除，这两个工具会自动清理旧版。

如果之前是自己动手安装的 Django，卸载也不麻烦，只需从 Python 的 `site-packages` 目录中删除 `django` 目录。为了找到需要删除的目录，可以在 shell（不是交互式 Python 控制台）中运行下述命令：

```
python -c "import sys; sys.path = sys.path[1:]; import django; print(django.__path__)"
```

20.4 安装针对特定发行版的包

若想检查你使用的平台/发行版有没有官方的 Django 包（安装程序），参阅 Django 文档中对各发行版的说明。针对特定发行版的包往往能自动安装依赖，而且便于升级。然而，这些包很少有 Django 的最新版。

20.5 安装开发版

如果决定使用 Django 的最新开发版，一定要紧盯开发时间线，还要关注下一个版本的发布记。使用开发版的好处是能第一时间使用新特性，而且能及早修改代码，应对新版本。（发布记中会详述各稳定版的重要变化。）

如果想偶尔升级 Django 代码，获取最近修正的缺陷和做出的改进，按照下述说明操作：

1. 确保系统中安装了 Git，而且可以在 shell 中使用 Git 命令。（在 shell 中输入 `git help`，测试可不可用。）
2. 检出 Django 的主开发分支（“trunk”或“master”），如下所示：

   ```
   git clone git://github.com/django/django.git django-trunk
   ```

   这个命令会在当前目录中创建 `django-trunk` 目录。
3. 确保 Python 解释器可以加载 Django 的代码。最便利的方式是使用 pip。执行下述命令：

   ```
   sudo pip install -e django-trunk/
   ```

   （如果使用 virtualenv 或者 Windows，可以省略 `sudo`。）这个命令的作用是让 Django 的代码可以导入，并让 `django-admin` 实用命令可用。至此结束！

提醒

不要运行 `sudo python setup.py install` 命令，因为第 3 步那个命令已经执行相关操作了。想升级 Django 源码时，只需在 `django-trunk` 目录中运行 `git pull` 命令，Git 将自动下载所有改动。

20.6 接下来

下一章详细说明如何使用不同的数据库运行 Django。

第 21 章 数据库管理进阶

本章详细说明如何在 Django 中使用各个支持的关系数据库，还会给出一些连接旧数据库的注意事项和小技巧。

21.1 通用说明

Django 已经尽力多支持各个数据库后端的功能，但是数据库后端之间是有差异的，因此 Django 开发者会有选择性地为特定的功能提供支持，而且会选择安全的方式实现。

本章说明在使用 Django 的过程中可能用到的一些功能。当然，说得再细致也不能代替各数据库后端的文档或参考手册。

21.1.1 持久连接

持久连接的消耗更小，无需每次请求都重新与数据库建立连接。持久连接的最长时间由 `CONN_MAX_AGE` 设置控制。各数据库可以分别设定 `CONN_MAX_AGE` 值。默认值为 `0`，即使用旧的行为，每次请求结束后断开连接。若想打开持久连接，把 `CONN_MAX_AGE` 设为一个表示秒数的正数。如果想一直持久连接，把值设为 `None`。

连接管理

首次查询数据库时，Django 连接到数据库。随后，保持连接，供后续查询复用。超过 `CONN_MAX_AGE` 定义的最大时长，或者不再需要使用时，Django 断开与数据库的连接。

具体而言，需要查询数据库而连接尚未建立时，不管是首次查询，还是之前的连接断开了，Django 会自动建立连接。

每次请求开始时，如果连接的时间超过了允许的最大时长，Django 会断开连接。如果连接空闲一段时间之后才被断开，应该把 `CONN_MAX_AGE` 的值设得小一点儿，以防 Django 使用已近被数据库服务器终止的连接。（这个问题可能只影响流量非常小的网站。）

每次请求结束时，如果连接超过了允许的最大时长，或者处于无法恢复的错误状态，Django 会将其断开。如果在处理请求的过程中出现了数据库错误，Django 会检查连接是否依然可用，如果不可用就将其断开。因此，数据库错误最多影响一个请求；一旦连接不可用，下一个请求会重新连接。

注意事项

因为每个线程都维护着一个连接，所以使用多少线程最少要同时有多少个连接。

有时，多数视图无法访问数据库，这可能是因为数据库在外部系统中，或者是缓存在起作用。遇到这种情况，应该把 `CONN_MAX_AGE` 设为较小的值，或者干脆设为 `0`，因为无法复用的连接无需再去维护了。这样，数据库的同时连接数也变小了。

开发服务器为要处理的每个请求新建一个线程，为的是避免持久连接。别在开发过程中建立持久连接。

Django 与数据库建立连接时，会根据所用的数据库后端设定适当的参数。如果使用持久连接，不会每次请求都重复设定参数。如果想修改参数，例如连接的隔离级别或时区，应该在每次请求结束后还原成 Django 的默

认值，迫使每次请求开始时使用恰当的值，或者禁用持久连接。

21.1.2 编码

Django 假定所有数据库都使用 UTF-8 编码。使用其他编码可能导致意料之外的行为，例如对 Django 有效的数据可能导致数据库报错，提醒值太长。正确设置数据库的方法参见后文对各个数据库的说明。

21.2 PostgreSQL 说明

Django 支持 PostgreSQL 9.0 及以上版本。为此，要使用 Psycopg2 2.0.9 或以上版本。

21.2.1 优化 PostgreSQL 的配置

连接数据库时，Django 需要下述参数：

- `client_encoding`：`'UTF8'`
- `default_transaction_isolation`：使用默认值 `'read committed'`，或者连接选项中设定的值（参见下文）。
- `timezone`：`USE_TZ` 设置为 `True` 时为 `'UTC'`，否则使用 `TIME_ZONE` 设置的值。

如果这些参数已经设为正确的值，Django 直接拿来用，不用在每次连接时重新设定，这样能稍微提升一点性能。这些参数可以在 `postgresql.conf` 文件中配置；如果想更细致，为各个数据库用户配置不同的值，使用 `ALTER ROLE`。

如果不设定这些参数，Django 也能与数据库连接，只不过每次连接时都要做额外的查询，设定这些参数。

21.2.2 隔离级别

与 PostgreSQL 自身一样，Django 默认使用的隔离级别是 `READ COMMITTED`。如果需要更高的隔离级别，例如 `REPEATABLE READ` 或 `SERIALIZABLE`，在 `DATABASES` 设置的 `OPTIONS` 选项中配置：

```
import psycopg2.extensions

DATABASES = {
    # ...
    'OPTIONS': {
        'isolation_level': psycopg2.extensions.ISOLATION_LEVEL_SERIALIZABLE,
    },
}
```

在较高的隔离级别下，应用程序要做好准备，处理序列化失败抛出的异常。这个选项针对高级用户。

21.2.3 索引 varchar 和 text 列

Django 通常为指定了 `db_index=True` 的模型字段输出一个 `CREATE INDEX` 语句。然而，如果数据库字段的类型是 `varchar` 或 `text`（例如 `CharField`、`FileField` 或 `TextField`），Django 会使用适当的 PostgreSQL 运算符类额外再为字段创建一个索引。为了能正确执行包含 `LIKE` 运算符的 SQL（即 `contains` 和 `startswith` 查询类型），必须要有那个额外的索引。

21.3 MySQL 说明

21.3.1 支持的版本

Django 支持 MySQL 5.5 及以上版本。

Django 的 `inspectdb` 功能使用 `information_schema` 数据库，这里包含关于所有数据库模式的详细数据。

Django 假定数据库支持 Unicode（UTF-8 编码），把事务和参照完整性都委托给数据库。值得注意的是，使用 MyISAM 存储引擎时，MySQL 并不强制实施事务和参照完整性（参见下一节）。

21.3.2 存储引擎

MySQL 有多种存储引擎。默认采用的存储引擎可以在服务器配置中修改。

在 MySQL 5.5.4 之前，默认的存储引擎是 MyISAM。这个存储引擎的主要缺点是不支持事务和外键约束。但它也有优点，在 MySQL 5.6.4 之前，它是唯一支持全文索引和搜索的引擎。

从 MySQL 5.5.5 开始，默认的存储引擎变成了 InnoDB。这个引擎完全支持事务和外键引用。目前，这个引擎可能是最好的选择。然而要注意，重启 MySQL 后，InnoDB 的自增计数器会丢失，因为它无法记住 `AUTO_INCREMENT` 的值，只能重建为 `max(id)+1`。这可能导致意外重用 `AutoField` 的值。

把现有项目升级到 MySQL 5.5.5 之后，如果新增了数据表，要确保所有表使用相同的存储引擎（即 MyISAM 或 InnoDB）。极为突出的是，如果是有 `ForeignKey` 的两个表，运行 `migrate` 命令时可能出现下述错误：

```
_mysql_exceptions.OperationalError: (
    1005, "Can't create table '\\db_name\\.#sql-4a8_ab' (errno: 150)"
)
```

21.3.3 MySQL DB API 驱动

Python Database API 在 PEP 249 中说明。有三个重要的 MySQL 驱动实现了这个 API：

- MySQLdb 是一个原生驱动，这十年来一直由 Andy Dustman 开发和提供支持。
- mysqlclient 是 MySQLdb 的派生版本，主要特色是支持 Python 3，而且可以直接替代 MySQLdb。截至目前，在 Django 中使用 MySQL 推荐使用这个驱动。
- MySQL Connector/Python 由 Oracle 推出，是使用纯 Python 实现的驱动，无需 MySQL 客户端库和 Python 标准库之外的模块。

这些驱动对线程都是安全的，而且都提供了连接池。目前，MySQLdb 是唯一不支持 Python 3 的驱动。

除了 DB API 驱动之外，为了让 Django 的 ORM 能访问数据库驱动，还需要*适配器*（adapter）。Django 为 MySQLdb 和 mysqlclient 提供了适配器，而 MySQL Connector/Python 自带了适配器。

mySQLdb

Django 要求使用 MySQLdb 1.2.1p2 或以上版本。

如果使用的过程中报错 `ImportError: cannot import name ImmutableSet`，说明你安装的 MySQLdb 中 `sets.py` 文件过时了，与 Python 2.4 及以上版本自带的同名内置模块有冲突。修正的方法是，确认安装的 MySQLdb 是 1.2.1p2 或以上版本，然后删除旧版在 MySQLdb 目录中遗留的 `sets.py` 文件。

MySQLdb 无法正确把日期字符串转换成 `datetime` 对象，这是已知的问题。具体而言，日期字符串 `0000-00-00` 对 MySQL 而言是有效的，但是 MySQLdb 会将其转换成 `None`。

也就是说，使用 loaddata/dumpdata 处理包含 `0000-00-00` 值的行时要小心，因为它们会被转换成 `None`。

截至目前，MySQLdb 的最新版本（1.2.4）还未支持 Python 3。若想在 Python 3 中使用 MySQLdb，只能安装 mysqlclient。

mysqlclient

Django 要求使用 mysqlclient 1.3.3 或以上版本。注意，mysqlclient 不支持 Python 3.2。除了对 Python 3.3+ 的支持，mysqlclient 的行为基本上与 MySQLdb 一致。

MySQL Connector/Python

MySQL Connector/Python 可从这个页面下载。1.1.X 及以上版本包含 Django 适配器。这个驱动可能不支持 Django 最近发布的版本。

21.3.4 时区定义

如果打算使用 Django 的时区功能，使用 `mysql_tzinfo_to_sql` 把时区数据表加载到 MySQL 数据库中。一个 MySQL 服务器只需做一次，不用每个数据库都加载。

21.3.5 创建数据库

可以在命令行工具中使用下述 SQL 创建数据库：

```
CREATE DATABASE <dbname> CHARACTER SET utf8;
```

这样所有表和列默认都将使用 UTF-8。

排序规则设置

列的*排序规则*（collation）控制着存储数据的顺序，以及把哪些字符串视为相等的。排序规则可以在整个数据库层设定，也可以在各个表和列上设定。MySQL 文档中有详细说明。不管在哪里设定，都要直接操纵数据库表，因为 Django 没有在模型定义中提供设定方式。

UTF-8 编码的数据库默认使用 `utf8_general_ci` 排序规则。在这个规则下，比较字符串相等性时不区分大小写。也就是说，数据库把 `"Fred"` 和 `"freD"` 视为相等的字符串。如果在一个字段上设定了唯一性约束，就无法在同一列中插入 `"aa"` 和 `"AA"`，因为在默认的排序规则下二者是相等的（也就不是唯一的）。

很多情况下使用这个默认值没有问题。然而，如果确实想在某列或某个表中使用区分大小写的比较方式，可以把列或表的排序规则改为 `utf8_bin`。注意，使用 MySQLdb 1.2.2 时，Django 的数据库后端从数据库的字符字段中获取的值都是字节字符串（而不是 Unicode 字符串）。这与 Django 始终返回 Unicode 字符串的做法完全不同。

如果让数据表使用 `utf8_bin` 排序规则，开发者要负责处理得到的字节字符串。多数情况下，Django 自身能顺利处理这样的列（不含 `contrib.sessions Session` 和 `contrib.admin LogEntry` 表，参见下文），但是你编写的代码也做好调用 `django.utils.encoding.smart_text()` 的准备，以防需要处理一致的数据，因为 Django 不会为你代劳（数据库后端层与模型处理层是分开的，因此遇到这种情况时数据库层不知道要转换）。

使用 MySQLdb 1.2.1p2 时，即便使用 `utf8_bin` 排序规则，Django 标准的 `CharField` 字段仍然返回 Unicode 字符串。然而，`TextField` 字段返回的是 `array.array` 实例（出自 Python 标准库中的 `array` 模块）。对此，

Django 也是无能为力，原因同上，从数据库中读取数据时没有足够的信息，无法转换。MySQLdb 1.2.2 修正了这个问题，所以，如果想在 `utf8_bin` 排序规则下使用 `TextField`，请升级到 1.2.2 版，然后按照前文所述的方式处理字节字符串（不是很难）。

使用 MySQLdb 1.2.1p2 或 1.2.2 时，你可能会让部分表使用 `utf8_bin` 排序规则，但是 `django.contrib.sessions.models.Session` 表（通常称为 `django_session`）和 `django.contrib.admin.models.LogEntry` 表（通常称为 `django_admin_log`）仍然要使用（默认的）`utf8_general_ci` 排序规则。根据 MySQL 文档中的“Unicode Character Sets”一文，与 `utf8_unicode_ci` 相比，在 `utf8_general_ci` 排序规则下比较的速度更快，但是准确性稍差一些。如果你的应用程序能接受这一点，应该使用 `utf8_general_ci`，因为它速度更快；如果不能接受（例如需要使用德语字典顺序），应该使用 `utf8_unicode_ci`，因为它更准确。

> **提醒**
>
> 模型表单集（formset）以区分大小写的方式验证唯一性字段。因此，使用不区分大小写的排序规则时，如果唯一性字段中的值只有大小写不同是能通过验证的，但是调用 `save()` 时会抛出 `IntegrityError` 异常。

21.3.6 连接数据库

连接设置按照下述顺序使用：

- `OPTIONS`
- `NAME`，`USER`，`PASSWORD`，`HOST`，`PORT`
- MySQL 选项文件

也就是说，如果在 `OPTIONS` 中设定了数据库的名称，它的优先级比 `NAME` 高，还会覆盖 MySQL 选项文件中的设置。下述示例配置使用 MySQL 选项文件：

```
# settings.py
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.mysql',
        'OPTIONS': {'read_default_file': '/path/to/my.cnf',},
    }
}

# my.cnf
[client]
database = NAME
user = USER
password = PASSWORD
default-character-set = utf8
```

可能用到的还有 MySQLdb 的其他连接选项，例如 `ssl`、`init_command` 和 `sql_mode`。详情参见 MySQLdb 文档。

21.3.7 创建数据表

Django 生成模式时不指定存储引擎，因此创建的表使用为数据库服务器配置的默认存储引擎。

所以，把数据库服务器的默认存储引擎设为想用的引擎是最简单的方法。

如果你使用的是托管服务，无法修改数据库服务器的默认存储引擎，有以下几个选择：

- 创建表之后执行 `ALTER TABLE` 语句，转换成新的存储引擎（例如 InnoDB）：

  ```
  ALTER TABLE <tablename> ENGINE=INNODB;
  ```

  如果有大量表，这么做很麻烦。

- 另一个选择是在创建表之前使用 MySQLdb 的 `init_command` 选项：

  ```
  'OPTIONS':{
      'init_command': 'SET storage_engine=INNODB',
  },
  ```

  这种方法在连接数据库时就设定默认存储引擎。创建表之后应该把这个选项删除，因为它增加了一个查询，而这个查询只在创建表时需要。

21.3.8 表名

MySQL（即便是最新版本）有个已知问题：在特定情况下执行特定的 SQL 语句可能导致表名的大小写发生变化。如果可能，建议表名使用小写字母，以防被这个行为影响。Django 根据模型自动生成的表名使用小写字母，因此通过 `db_table` 参数覆盖表名时要注意这一点。

21.3.9 保存点

Django ORM 和 MySQL（使用 InnoDB 存储引擎时）都支持数据库*保存点*（savepoint）。

使用 MyISAM 存储引擎时要注意，事务 API 中与保存点有关的方法可能导致数据库出错。这是因为检测 MySQL 数据库（表）使用哪种存储引擎是个耗费资源的操作，因此判定在检测结果中不值得动态转换这些方法。

21.3.10 某些字段要注意的事项

字符字段

对列类型为 `VARCHAR` 的字段来说，如果为字段设定了 `unique=True`，那么字段中最多能存储 255 个字符。受影响的有 `CharField`、`SlugField` 和 `CommaSeparatedIntegerField`。

时间和日期时间字段对微秒的支持

MySQL 5.6.4 及以上版本能存储微秒，前提是定义列时指定包含微秒（例如 `DATETIME(6)`）。之前的版本完全不支持。此外，MySQLdb 1.2.5 之前的版本有个缺陷，无法正确在 MySQL 中存储微秒。

即使数据库服务器支持微秒，Django 也不会自动更新现有列，包含微秒。如果想在现有数据库中存储微秒，要执行下述命令，自己动手更新列：

```
ALTER TABLE `your_table` MODIFY `your_datetime_column` DATETIME(6)
```

也可以在数据迁移中执行 `RunSQL` 操作。

使用 MySQL 5.6.4 或以上版本，并且使用 mysqlclient 或 MySQLdb 1.2.5 或以上版本时，新建的 `DateTimeField` 或 `TimeField` 列默认包含微秒。

时间戳列

如果使用旧数据库，包含 TIMESTAMP 列，必须设定 USE_TZ = False，以防数据损坏。inspectdb 命令会把这种类型的列映射到 DateTimeField 上；如果启用了时区支持，MySQL 和 Django 都会尝试把时间戳值从 UTC 转换成本地时间。

使用 Queryset.select_for_update() 锁定行

MySQL 不支持在 SELECT … FOR UPDATE 语句中使用 NOWAIT 选项。如果设定了 nowait=True，调用 select_for_update() 时会抛出 DatabaseError 异常。

自动转换类型可能导致意外结果

查询类型为字符串但实际存储整数值的列时，在比较之前 MySQL 会把表中所有类型的值强制转换成整数。如果表中包含 "abc" 或 "def" 这样的值，而查询条件是 WHERE mycolumn=0，这两行都将匹配。类似地，WHERE mycolumn=1 将匹配 "abc1"。因此，在 Django 中定义的字符串类型字段，在查询中使用之前始终会被转换成字符串。

如果你自定义的模型字段直接继承自 Field，而且覆盖了 get_prep_value()，或者使用 extra() 或 raw()，应该确保做了恰当的类型转换。

21.4 SQLite 说明

如果应用程序主要用于读取数据，或者不想安装大量支持工具，在开发过程中特别适合使用 SQLite。不过数据库服务器之间存在着差异，有些 SQLite 特有的差异是你需要知晓的。

21.4.1 子串匹配和大小写

所有 SQLite 版本在匹配某些类型的字符串上都有一些违背直觉的行为。这些行为在查询集合上使用 iexact 或 contains 过滤器时便能体现出来。这些行为分为两种情况：

1. 匹配子串时不区分大小写。例如 name__contains="aa" 过滤器能匹配 "Aabb"。
2. 精确匹配包含 ASCII 以外字符的字符串时，即便在查询中指定了不区分大小写的选项，仍然以区分大小写的方式匹配。因此，在这种情况下，iexact 过滤器的行为与 exact 过滤器完全一样。

SQLite 网站中给出了一些变通方案，但是 Django 默认的 SQLite 后端并未采用，因为那些方案很难正确集成。因此，Django 提供的是 SQLite 的默认行为。想以不区分大小写的方式过滤，或者过滤子串时要知晓这一点。

21.4.2 旧版 SQLite 和 CASE 表达式

SQLite 3.6.23.1 及以下版本在处理包含 ELSE 和算术运算的 CASE 表达式中的查询参数时有个缺陷。

SQLite 3.6.23.1 在 2010 年 3 月发布，而针对各平台的二进制分发版本如今大都包含较新的版本，不过 Windows 的 Python 2.7 安装程序是个例外。

截至目前，针对 Windows 的最新版本（Python 2.7.10）包含 SQLite 3.6.21。这个问题的补救措施是安装 pysqlite2，或者从 SQLite 网站中下载较新版本的 sqlite3.dll，把旧版替换掉（默认安装到 C:\Python27\DLLs 文件夹中）。

21.4.3 使用 SQLite DB-API 2.0 驱动的较新版本

如果安装有 `pysqlite2`，Django 将优先使用，替代 Python 标准库中自带的 `sqlite3`。

这样，如果需要，可以把 DB-API 2.0 接口或 SQLite 3 自身升级到比 Python 二进制分发文件中自带的更新的版本。

21.4.4 数据库锁定错误

SQLite 是轻量级数据库，不支持大量并发。如果出现 `OperationalError: database is locked` 错误，说明应用程序的并发数超过了 SQLite 默认配置所能处理的数量。这个错误表明某个线程或进程在数据库连接上施加了排它锁，其他线程在等待锁释放的过程中超时了。

Python 的 SQLite 包装程序为线程等待锁释放设定了默认的超时时间，超过这个值就抛出 `OperationalError: database is locked` 错误。

这个错误可以采用下述方法解决：

- 换用其他数据库后端。在特定的时刻，SQLite 对真实的应用程序来说太过轻量了；并发错误就表明到了这样的时刻。
- 重写代码，减少并发，并且确保数据库事务历时较短。
- 在数据库选项中设定 `timeout`，设为较大的超时时间：

```
'OPTIONS':{
    # ...
    'timeout': 20,
    # ...
},
```

这样只是等待时间变长了，对锁定错误本身没有采取任何解决措施。

21.4.5 不支持 `QuerySet.select_for_update()`

SQLite 不支持 `SELECT … FOR UPDATE` 句法。即便调用也没任何效果。

21.4.6 原始查询不支持 Python 那种指定参数的格式

多数后端的原始查询（`Manager.raw()` 或 `cursor.execute()`）可以使用 Python 那种指定参数的格式，即在查询中以 `'%(name)s'` 形式指定占位符，然后通过字典（而非列表）传入参数。SQLite 不支持这种方式。

21.4.7 `connection.queries` 中放在引号里的参数

如果把参数放在引号中代换，`sqlite3` 无法获取 SQL。`connection.queries` 输出的 SQL 是通过字符串插值重新构建的，可能不准确。把查询复制到 SQLite shell 之前一定要添加必要的引号。

21.5 Oracle 说明

Django 支持 Oracle Database Server 11.1 及以上版本。`cx_Oracle` Python 驱动要使用 4.3.1 或以上版本，不过推荐使用 5.1.3 或以上版本，因为这些版本支持 Python 3。

注意，cx_Oracle 5.0 有个缺陷，会损坏 Unicode，因此不要在 Django 中使用。cx_Oracle 5.0.1 修正了这个问题，因此如果想使用较新的版本，使用 5.0.1 版吧。

编译 cx_Oracle 5.0.1 或以上版本时可以选择设定 WITH_UNICODE 环境变量。推荐设定这个环境变量，但不强制。

为了让 python manage.py migrate 能正常使用，你的 Oracle 数据库用户必须具有执行下述操作的权限：

- CREATE TABLE
- CREATE SEQUENCE
- CREATE PROCEDURE
- CREATE TRIGGER

为了运行项目的测试组件，数据库用户通常还需要具有下述额外的权限：

- CREATE USER
- DROP USER
- CREATE TABLESPACE
- DROP TABLESPACE
- CREATE SESSION WITH ADMIN OPTION
- CREATE TABLE WITH ADMIN OPTION
- CREATE SEQUENCE WITH ADMIN OPTION
- CREATE PROCEDURE WITH ADMIN OPTION
- CREATE TRIGGER WITH ADMIN OPTION

注意，虽然 RESOURCE 角色具有必须的 CREATE TABLE、CREATE SEQUENCE、CREATE PROCEDURE 和 CREATE TRIGGER 权限，而且具有 RESOURCE WITH ADMIN OPTION 权限的用户可以把 RESOURCE 角色赋予他人，但是这样的用户不能把单独的权限（如 CREATE TABLE）赋予他人，因此 RESOURCE WITH ADMIN OPTION 权限通常还不足以运行测试。

有些测试组件还要创建视图，为此，用户还要具有 CREATE VIEW WITH ADMIN OPTION 权限。Django 自己的测试组件就是如此。

这些权限 DBA 角色都具有。在开发者个人私有的数据库中适合使用这个角色。

Oracle 数据库后端使用 SYS.DBMS_LOB 包，因此用户要有在其上执行操作的权限。通常，所有用户都有这个权限，但也有例外；如果没有这个权限，要像下面这样赋予：

```
GRANT EXECUTE ON SYS.DBMS_LOB TO user;
```

21.5.1 连接数据库

如果使用 Oracle 数据库的服务名连接数据库，在 settings.py 文件中要像下面这样设置：

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.oracle',
        'NAME': 'xe',
        'USER': 'a_user',
```

```
            'PASSWORD': 'a_password',
            'HOST': '',
            'PORT': '',
        }
    }
```

此时，HOST 和 PORT 应该留空。然而，如果不使用 tnsnames.ora 文件或类似的命名方法，而想使用 SID（下例中的 xe）连接数据库，要像下面这样设定 HOST 和 PORT：

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.oracle',
        'NAME': 'xe',
        'USER': 'a_user',
        'PASSWORD': 'a_password',
        'HOST': 'dbprod01ned.mycompany.com',
        'PORT': '1540',
    }
}
```

HOST 和 PORT 要么留空，要么都设定。Django 会根据这两个选项的设定情况选择不同的连接描述符（descriptor）。

21.5.2 threaded 选项

如果计划在多线程环境中运行 Django（例如在现代的操作系统中使用默认的 MPM 模块运行 Apache），必须在 Oracle 数据库的配置中把 threaded 选项设为 True：

```
'OPTIONS': {
    'threaded': True,
},
```

如若不然，可能导致崩溃和其他怪异的行为。

21.5.3 INSERT … RETURNING INTO

Oracle 后端插入新行时，默认使用 RETURNING INTO 子句高效获取 AutoField 的值。在某些特别的情况下，这种行为可能导致 DatabaseError，例如插入远程表，或者插入设定了 INSTEAD OF 触发器的视图。

若想禁用 RETURNING INTO 子句，在数据库配置中把 use_returning_into 选项设为 False：

```
'OPTIONS': {
    'use_returning_into': False,
},
```

此时，Oracle 后端将使用单独的一个 SELECT 查询获取 AutoField 的值。

21.5.4 命名问题

Oracle 限制名称的最大长度为 30 个字符。

鉴于此，后端会截断数据库标识符，把截断后的名称的后四个字符换成可复现的 MD5 哈希值。此外，后端会把数据库标识符变成全大写。

为了防止这些转换（通常只在处理旧数据库或访问别人的表时才有这样的要求），把名称放在引号里：

```
class LegacyModel(models.Model):
    class Meta:
        db_table = '"name_left_in_lowercase"'

class ForeignModel(models.Model):
    class Meta:
        db_table = '"OTHER_USER"."NAME_ONLY_SEEMS_OVER_30"'
```

放在引号中的名称可以在 Django 支持的其他数据库后端中使用，但唯独 Oracle 不支持。

如果使用 Oracle 的关键字命名模型字段或者作为 `db_column` 选项的值，运行 `migrate` 命令时可能会遇到 `ORA-06552` 错误。Django 把查询中用到的所有标识符都放在引号里，这样在多数情况下能避免这种问题，但是用 Oracle 的数据类型命名列时依然可能会遇到这个错误。因此，尤其要小心，别把字段命名为 `date`、`timestamp`、`number` 或 `float`。

21.5.5 NULL 和空字符串

Django 一般倾向于使用空字符串（''），而不是 `NULL`，但是在 Oracle 看来，二者是等价的。为了解决这个问题，Oracle 后端忽略允许使用空字符串的字段上设定的 `null` 选项，按照 `null=True` 生成 DDL。从数据库中读取数据时，Oracle 后端假定这样的字段中存储的 `NULL` 值其实是表示空字符串，并且据此悄无声息地转换数据。

21.5.6 对 TextField 的限制

Oracle 后端把 `TextField` 存储为 `NCLOB` 类型。此时，Oracle 会对 LOB 类型的列做出一些限制：

- LOB 列不可以用作主键。
- LOB 列不能建立索引。
- LOB 列不能使用 `SELECT DISTINCT` 查询。这意味着，使用 Oracle 时，在包含 `TextField` 字段的模型上调用 `QuerySet.distinct` 会出错。解决方法是，结合 `QuerySet.defer` 和 `distinct()` 方法，把 `TextField` 字段排除在 `SELECT DISTINCT` 得到的列表之外。

21.6 使用第三方数据库后端

除了官方支持的数据库之外，还有些第三方后端，让你能在 Django 中使用其他数据库：

- SAP SQL Anywhere
- IBM DB2
- Microsoft SQL Server
- Firebird
- ODBC
- ADSDB

这些非官方后端支持的 Django 版本和 ORM 功能差异巨大。关于这些非官方后端具体支持的功能和查询类型，请通过各项目的支持渠道询问。

21.7 集成旧数据库

虽然 Django 最适合用来开发新应用程序，但是也不是不能集成旧数据库。Django 自带了很多实用工具，力求简化这一过程。

安装好 Django 之后，可以按照下述一般过程集成现有数据库。

21.7.1 配置数据库参数

你要告诉 Django 数据库连接参数和数据库名。在 `DATABASES` 设置中为 `'default'` 连接的下述各个键设值：

- `NAME`
- `ENGINE <DATABASE-ENGINE>`
- `USER`
- `PASSWORD`
- `HOST`
- `PORT`

21.7.2 自动生成模型

Django 自带的 `inspectdb` 实用工具可以通过内省现有数据库创建模型。运行下述命令可以看到相关输出：

```
python manage.py inspectdb
```

使用标准的 Unix 输出重定向把输出的内容保存到一个文件中（在 Windows 中也可以使用）：

```
python manage.py inspectdb > models.py
```

这个工具的目的只是为了节省时间，不能作为生成模型的最终结果。详情参见 `inspectdb` 的文档。

整理好模型之后，把文件命名为 `models.py`，保存到应用所在的 Python 包里。然后把应用添加到 `INSTALLED_APPS` 设置中。

`inspectdb` 默认创建不用管理的模型。即在模型的 `Meta` 类中设有 `managed = False`，不让 Django 负责创建、修改和删除表。

```
class Person(models.Model):
    id = models.IntegerField(primary_key=True)
    first_name = models.CharField(max_length=70)
    class Meta:
        managed = False
        db_table = 'CENSUS_PERSONS'
```

如果想让 Django 管理表的生命周期，要把 `managed` 选项的值改为 `True`（或者干脆删除，因为 `True` 是默认值）。

21.7.3 安装 Django 的核心表

接下来，运行 `migrate` 命令，安装其他所需的数据库记录，例如管理权限和内容类型：

```
python manage.py migrate
```

21.7.4 清理生成的模型

正如你想，数据库内省不完美，得到的模型代码要适当做些清理。下面给出几点提示：

- 每个数据库表转换成一个模型类（即数据库表与模型类之间是一对一关系）。因此，要重构多对多联结表的模型，改成 `ManyToManyField` 对象。
- 生成的各个模型中包含每一个字段，包括 ID 主键字段。然而，Django 在没有 ID 主键的情况下会自动添加。因此，要把类似这样的代码删除：

  ```
  id = models.IntegerField(primary_key=True)
  ```

  之所以删除，不仅是因为多余，如果要在表中添加新记录，存在这个字段还可能导致问题。
- 字段的类型（如 `CharField`、`DateField`）通过数据库列的类型（如 `VARCHAR`、`DATE`）确定。如果 `inspectdb` 无法判断，将使用 `TextField`，并在生成的字段代码旁插入一个注释：`'This field type is a guess.'`（这个字段的类型是猜的）。留意这样的注释，如有必要，相应地修改字段类型。
- 如果数据库中的列在 Django 中没有适当的字段对应，可以放心将其忽略。Django 模型层无需包含表中的每个列。
- 如果数据库列的名称是 Python 保留字（如 `pass`、`class`、`for`），`inspectdb` 将在属性名后添加 `"_field"`，并把 `db_column` 参数设为真正的字段名（如 `pass`、`class`、`for`）。假如有个名为 `for` 的 `INT` 类型列，生成的模型会像下面这样定义：

  ```
  for_field = models.IntegerField(db_column='for')
  ```

  `inspectdb` 会在这样的字段旁插入一个注释：`'Field renamed because it was a Python reserved word.'`（字段重命名了，因为它是 Python 保留字）。
- 如果数据库中有引用其他表的表（多数数据库都有），可能要调整生成的模型顺序，以正确的顺序排列引用其他模型的模型。假如 `Book` 模型有个外键引用 `Author` 模型，那么 `Author` 模型就要在 `Book` 模型前面定义。如果要建立关联的模型尚未定义，可以用字符串表示模型的名称，不能直接使用模型对象。
- `inspectdb` 能检测出 PostgreSQL、MySQL 和 SQLite 的主键，即能适时插入 `primary_key=True`。其他数据库则要自己动手，至少在模型中的一个字段上设定 `primary_key=True`，因为这是 Django 模型的要求。
- 只有 PostgreSQL 和特定类型的 MySQL 表才能检测出外键。检测不出来时，外键以 `IntegerField` 表示，即假定外键是 `INT` 类型的列。

21.7.5 测试，微调

以上是基本步骤，做完之后还要微调 Django 生成的模型，直到符合自己的要求。试着通过 Django 数据库 API 访问数据、在 Django 管理后台中编辑对象，然后相应地修改模型文件。

21.8 接下来

本书正文到此结束！

希望本书没有让你失望，读完之后能有所获益。虽然本书可以作为 Django 全面的参考手册，但是没有什么能取代动手操作。代码敲起来吧，望你借助 Django 取得一番成就！

余下的几篇附录详解 Django 中的各个函数和字段，仅作参考之用。

附录 A 模型定义参考指南

第 4 章讲解了模型定义的基础知识，为后文奠定了基础。然而，还有大量可用的模型选项没有涵盖。本篇附录说明模型定义可用的各个选项。

A.1 字段

模型最重要的部分（也是模型唯一必须的部分）是所定义的数据库字段。

A.1.1 对字段名称的限制

Django 对模型字段的名称做了两个限制：

1. 字段名称不能是 Python 保留字，否则会导致 Python 句法出错。例如：

    ```
    class Example(models.Model):
        pass = models.IntegerField() # “pass”是保留字！
    ```

2. 字段名称不能包含多个连续的下划线，否则会影响 Django 的查询查找句法。例如：

    ```
    class Example(models.Model):
        # “foo__bar”中有两个下划线！
        foo__bar = models.IntegerField()
    ```

模型中的各个字段应该是适当 `Field` 类的实例。Django 通过字段所属的类确定以下几件事：

- 数据库列的类型（如 `INTEGER`、`VARCHAR`）。
- 在 Django 的表单和管理后台中使用的小组件（widget）（如 `<input type="text">`、`<select>`）。
- 必要的验证，供 Django 的管理界面和表单使用。

每个字段类都接受一组选项参数。例如，第 4 章中的 `book` 模型是这样定义 `num_pages` 字段的：

```
num_pages = models.IntegerField(blank=True, null=True)
```

这里，为字段类设置了 `blank` 和 `null` 选项。Django 中可用的各个字段选项参见表 A-2。

此外，有些类还定义了专门的选项。例如，`CharField` 有个必须的选项 `max_length`，其默认值为 `None`，如下所示：

```
title = models.CharField(max_length=100)
```

这里，我们把 `max_length` 字段选项设为 100，限制书名的长度不能超过 100 个字符。

表 A-1 按字母顺序列出了所有字段类。

表 A-1：Django 模型字段类

字段类	默认小组件	说明
AutoField	N/A	根据 ID 自动递增的 IntegerField。
BigIntegerField	NumberInput	64 位整数，与 IntegerField 很像，但取值范围是 -9223372036854775808 到 9223372036854775807。
BinaryField	N/A	存储原始二进制数据的字段。只支持 bytes 类型。注意，这个字段的功能有限。
BooleanField	CheckboxInput	真假值字段。如果想接受 null 值，使用 NullBooleanField。
CharField	TextInput	字符串字段，针对长度较小的字符串。大量文本应该使用 TextField。有个额外的必须参数：max_length，即字段的最大长度（字符个数）。
DateField	DateInput	日期，在 Python 中使用 datetime.date 实例表示。有两个额外的可选参数：auto_now，每次保存对象时自动设为当前日期；auto_now_add，创建对象时自动设为当前日期。
DateTimeField	DateTimeInput	日期和时间，在 Python 中使用 datetime.datetime 实例表示。与 DateField 具有相同的额外参数。
DecimalField	TextInput	固定精度的小数，在 Python 中使用 Decimal 实例表示。有两个必须的参数：max_digits 和 decimal_places。
DurationField	TextInput	存储时间跨度，在 Python 中使用 timedelta 表示。
EmailField	TextInput	一种 CharField，使用 EmailValidator 验证输入。max_length 的默认值为 254。
FileField	ClearableFileInput	文件上传字段。详情参见下一节。
FilePathField	Select	一种 CharField，限定只能在文件系统中的特定目录里选择文件。
FloatField	NumberInput	浮点数，在 Python 中使用 float 实例表示。注意，field.localize 的值为 False 时，默认的小组件是 TextInput。
ImageField	ClearableFileInput	所有属性和方法都继承自 FileField，此外验证上传的对象是不是有效的图像。增加了 height 和 width 两个属性。需要 Pillow 库支持。
IntegerField	NumberInput	整数。取值范围是 -2147483648 到 2147483647，在 Django 支持的所有数据库中可放心使用。
GenericIPAddressField	TextInput	IPv4 或 IPv6 地址，字符串形式（如 192.0.2.30、2a02:42fe::4）。

（续）

字段类	默认小组件	说明
NullBooleanField	NullBooleanSelect	类似于 BooleanField，但是 NULL 可作为其中一个选项。
PositiveIntegerField	NumberInput	整数。取值范围是 0 到 2147483647，在 Django 支持的所有数据库中可放心使用。
SlugField	TextInput	*别名*（slug）是报业术语，是某个事物的简短标注，只包含字母、数字、下划线或连字符。
SmallIntegerField	NumberInput	类似于 IntegerField，但是对值有限制。取值范围是 -32768 到 32767，在 Django 支持的所有数据库中可放心使用。
TextField	Textarea	大段文本字段。如果指定了 max_length 选项，这一限制在自动生成的表单字段中会体现出来。
TimeField	TextInput	时间，在 Python 中使用 datetime.time 实例表示。
URLField	URLInput	用于输入 URL 的 CharField。可选 max_length 选项。
UUIDField	TextInput	用于存储通用唯一标识码。使用 Python 的 UUID 类。

A.1.2 FileField 说明

不支持 primary_key 和 unique 选项，否则会抛出 TypeError。

有两个可选的参数：

1. FileField.upload_to
2. FileField.storage

FileField.upload_to

追加到 MEDIA_ROOT 设置后面的本地文件系统路径，用于构建 url 属性的值。可以包含 strftime() 格式化字符串，上传文件时将替换成当时的日期/时间（以防上传的文件填满目录）。还可以是可调用的对象，如函数，调用后返回上传的路径（包括文件名）。这个可调用对象必须接受两个参数，而且返回 Unix 风格的路径（使用正斜线），传给存储系统。接受的两个参数是：

- 实例。FileField 所在模型的实例。严格来说，是所上传文件依附的实例。多数情况下，这个对象尚未存入数据库，所以如果主键使用默认的 AutoField，主键字段可能还没有值。
- 文件名。文件原来的名称。最终得到的路径可能采用、也可能不采用这个文件名。

FileField.storage

存储器对象，处理文件的存取。这个字段默认的表单小组件是 ClearableFileInput。在模型中使用 FileField 或 ImageField（参见后文）要按照下述步骤来做：

- 在设置文件中把 MEDIA_ROOT 设为 Django 存储上传文件的目录的完整路径。（出于性能上的考虑，上传的文件不存储在数据库中。）把 MEDIA_URL 设为那个目录的公开 URL。运行 Web 服务器的用户账户必须有那个目录的写权限。
- 在模型中添加 FileField 或 ImageField，把 upload_to 选项设为 MEDIA_ROOT 中的一个子目录，用于存储上传的文件。
- 存储在数据库中的只是文件的路径（相对于 MEDIA_ROOT）。多数时候只需使用 Django 提供的 url 属性。例如，如果 ImageField 名为 mug_shot，在模板中可以使用 {{ object.mug_shot.url }} 获取图像的绝对路径。

上传文件时一定要谨慎处理文件的类型，以防出现安全漏洞。上传的所有文件都要验证，确保文件类型符合要求。倘若毫无防备，允许把文件上传到 Web 服务器文档根目录中的某个目录中，而不验证，不怀好意的人就可能上传 CGI 或 PHP 脚本，然后访问网站的某个 URL 执行脚本。可别犯傻！

还要注意，即便上传的是 HTML 文件也可能造成安全威胁，因为 HTML 文件会由浏览器执行（而非服务器），从而导致 XSS 或 CSRF 攻击。在数据库中，FileField 实例对应的是 varchar 列，最大长度默认为 100 个字符。与其他字段一样，可以通过 max_length 选项修改最大长度。

FileField 和 FieldFile

在模型上访问 FileField 时，是以 FieldFile 实例为代理访问底层文件的。除了继承自 django.core.files.File 的功能之外，这个类还有几个属性和方法，可与文件数据交互。

FieldFile.url

只读特性（property），调用底层 Storage 类的 url() 方法获取文件的相对 URL。

FieldFile.open(mode='rb')

行为与 Python 标准的 open() 方法类似，以 mode 指定的模式打开模型中与这个实例关联的文件。

FieldFile.close()

行为与 Python 标准的 file.close() 方法类似，关闭与这个实例关联的文件。

FieldFile.save(name, content, save=True)

这个方法把传入的文件名和文件内容传给字段的存储类，然后把存储的文件与模型字段关联起来。如果想自己动手把文件数据与模型中的 FileField 实例关联起来，使用 save() 方法持久存储文件数据。

这个方法有两个必须的参数：name 是文件的名称，content 是包含文件内容的对象。可选的 save 参数控制改动与字段关联的文件后是否保存模型实例。默认为 True。

注意，content 参数的值应该是 django.core.files.File 实例，而非 Python 内置的文件对象。可以像下面这样使用现有的 Python 文件对象构建 File 实例：

```
from django.core.files import File
# 使用 Python 内置的 open() 打开现有文件
f = open('/tmp/hello.world')
myfile = File(f)
```

也可以像这样使用 Python 字符串构建：

```
from django.core.files.base import ContentFile
myfile = ContentFile("hello world")
```

`FieldFile.delete(save=True)`

删除与这个实例关联的文件，并且清除字段上的所有属性。如果调用 `delete()` 时文件是打开的，还会将其关闭。

可选的 `save` 参数控制删除与字段关联的文件后是否保存模型实例。默认为 `True`。

注意，删除模型后，相关的文件不会删除。如果想清理无主文件，只能自己动手（例如，手动或通过 cron 任务定期运行一个自定义的管理命令）。

A.2 通用字段选项

表 A-2列出 Django 中所有可选的字段选项。这些选项可以在任何字段类型中使用。

表 A-2：Django 通用字段选项

选项	说明
`null`	设为 `True` 时，Django 在数据库中把空值存储为 `NULL`。默认为 `False`。基于字符串的字段，如 `CharField` 和 `TextField`，不应该使用 `null`，因为空字符串值始终存储为空字符串，而非 `NULL`。对基于字符串和不基于字符串的字段来说，如果想让表单接受空值，还要设定 `blank=True`。如果想让 `BooleanField` 接受 `null` 值，使用 `NullBooleanField`。
`blank`	设为 `True` 时，字段允许空白值。默认为 `False`。注意，这与 `null` 不同。`null` 只针对数据库，而 `blank` 是针对数据验证的。
`choices`	可迭代的对象（如列表或元组），由两个元组（包括自身）组成的可迭代对象构成（如 `[(A, B), (A, B) …]`），用于设定字段的选项。如果设定这个选项，默认的表单小组件将由标准的文本字段变成带选项的选择框。各元组中的第一个元素是真正在模型上设定的值，第二个元素是人类可读的名称。
`db_column`	字段使用的数据库列名称。如未指定，Django 将使用字段的名称。
`db_index`	设为 `True` 时，在字段上建立数据库索引。
`db_tablespace`	为有索引的字段指定索引使用的数据库*表空间*（tablespace）名称。默认为项目的 `DEFAULT_INDEX_TABLESPACE` 设置，或者模型的 `db_tablespace` 属性。如果数据库后端不支持为索引指定表空间，忽略这个选项。
`default`	字段的默认值。可以是一个值，也可以是一个可调用对象。为后者时，每次新建对象都会调用一次。默认值不能是*可变的*（mutable）对象（模型实例、列表、集，等等），因为对那个对象的引用将作为所有新模型实例中字段的默认值。
`editable`	设为 `False` 时，字段不在管理后台或其他 `ModelForm` 中显示。验证模型时也会跳过。默认为 `True`。

（续）

选项	说明
error_messages	用于覆盖字段抛出异常时的默认消息。值为一个字典，通过键指定想覆盖的错误消息。错误消息键包括 null、blank、invalid、invalid_choice、unique 和 unique_for_date。
help_text	在表单小组件旁显示的额外帮助文本。即便不在表单中显示，也能用作文档。注意，在自动生成的表单中，不会转义这里的 HTML，因此，如果需要，可以在帮助文本中使用 HTML。
primary_key	设为 True 时，指定字段为模型的主键。如果模型中没有一个字段设定 primary_key=True，Django 会自动添加一个 AutoField，用于存储主键，因此，除非想覆盖默认的主键行为，否则无需在任何字段上设定 primary_key=True。主键字段是只读的。
unique	设为 True 时，在表中字段的值必须是唯一的。这一限制由数据库层和模型验证实施。除了 ManyToManyField、OneToOneField 和 FileField 之外，其他字段都可以设定这个选项。
unique_for_date	设为 DateField 或 DateTimeField 字段的名称，确保与所在字段的组合是唯一的。假如有个 title 字段设定了 unique_for_date="pub_date"，那么 Django 不允许出现 title 和 pub_date 都相同的两条记录。这个限制在验证模型时由 Model.validate_unique() 实施，而不在数据库层实施。
unique_for_month	类似于 unique_for_date，不过验证唯一性时考虑的是月份。
unique_for_year	类似于 unique_for_date，不过验证唯一性时考虑的是年份。
verbose_name	字段的人类可读名称。如果未设定，Django 将使用字段的属性名称（下划线转换成空格）自动生成一个。
validators	用于验证字段的验证器列表。

A.3 字段属性参考指南

每个字段实例都有几个可用于内省行为的属性。如果想根据字段的功能编写代码，请使用这些属性，别通过 isinstance 检查。这些属性可以与 Model._meta API 结合起来进一步缩窄要查找的字段类型。自定义的模型字段应该实现下述旗标。

A.3.1 字段的属性

Field.auto_created

布尔旗标，指明字段是否为自动创建的。模型继承使用的 OneToOneField 就是自动创建的。

Field.concrete

布尔旗标，指明字段是否关联数据库列。

Field.hidden

布尔旗标，指明一个字段是否用于支持另一个非隐藏字段的功能（例如组成 GenericForeignKey 的 con-

tent_type 和 object_id 字段）。这个旗标用于把模型中公开的字段子集从全体字段中区分出来。

Field.is_relation

布尔旗标，指明一个字段是否包含指向另一个或多个模型的引用（如 ForeignKey、ManyToManyField、OneToOneField，等等）。

Field.model

返回定义字段的模型。如果字段在模型的超类中定义，model 指代超类，而不是所在模型类的实例。

A.3.2 包含关系的字段的属性

这些属性用于查询关系的基数和其他细节。所有字段都有这些属性，然而仅在关系类型的字段（Field.is_relation=True）上有意义。

Field.many_to_many

布尔旗标，字段有多对多关系时为 True，否则为 False。在 Django 中，只有 ManyToManyField 的这个属性值为 True。

Field.many_to_one

布尔旗标，字段有多对一关系时为 True，否则为 False。

Field.one_to_many

布尔旗标，字段有一对多关系时为 True（例如 GenericRelation 或 ForeignKey 的另一端），否则为 False。

Field.one_to_one

布尔旗标，字段有一对一关系时为 True（如 OneToOneField），否则为 False。

Field.related_model

指向与字段关联的那个模型。例如 ForeignKey(Author) 中的 Author。如果字段是通用关系（例如 GenericForeignKey 或 GenericRelation），那么 related_model 值为 None。

A.4 关系

Django 还定义了一系列表示关系的字段。

A.4.1 ForeignKey

多对一关系。有一个必须的位置参数：与模型关联的类。若想创建递归关系，即一个对象与自身有多对一关系，使用 models.ForeignKey('self')。

如果想与尚未定义的模型建立关系，不能使用模型对象，应该使用模型的名称：

```
from django.db import models
```

```
class Car(models.Model):
    manufacturer = models.ForeignKey('Manufacturer')
    # ...

class Manufacturer(models.Model):
    # ...
    pass
```

若想引用另一个应用中定义的模型，可以通过完整的应用标注明确指定模型。假如上述 Manufacturer 模型在名为 production 的应用中定义，可以这么做：

```
class Car(models.Model):
    manufacturer = models.ForeignKey('production.Manufacturer')
```

这样可以避免循环导入依赖。ForeignKey 上自动建立数据库索引。若想禁用，把 db_index 设为 False。

如果创建外键是为了保持一致性，而不是想做联结查询，或者是想以别的方式创建索引（例如部分索引或多列索引），可以禁止自动创建索引，避免开销。

数据库表述

Django 在背后会为字段名称添加 "_id" 后缀，作为数据库列的名称。在上述示例中，Car 模型对应的数据表中将出现 manufacturer_id 列。

这个行为可以通过 db_column 选项改变，然而，除非需要自己编写 SQL，否则不应该直接自定义数据库列名。我们处理的应该是模型对象的字段名称。

参数

ForeignKey 接受一些额外的参数（全为可选的），用于定义关系的细节。

limit_choices_to

使用 ModelForm 渲染或在管理后台中显示时限制罗列这个字段的条件（默认为查询集中的所有对象）。值为一个字典、一个 Q 对象，抑或返回一个字典或 Q 对象的可调用对象。例如：

```
staff_member = models.ForeignKey(User, limit_choices_to={'is_staff': True})
```

此时，ModelForm 中的相应字段只会列出 is_staff=True 的 User 对象。这对 Django 的管理后台可能有用。结合 Python 的 datetime 模块限制选择的日期范围时可以使用可调用对象。例如：

```
def limit_pub_date_choices():
    return {'pub_date__lte': datetime.date.utcnow()}

limit_choices_to = limit_pub_date_choices
```

如果 limit_choices_to 的值是或者返回一个 Q 对象（适用于复杂的查询），仅当模型的 ModelAdmin 子类的 raw_id_fields 属性列出字段时才对字段在管理后台中的选择有效果。

related_name

用于获取所关联对象的名称，也是 related_query_name（目标模型使用的反向过滤器名称）的默认值。详细说明和示例参见“反向”一节。注意，在抽象模型上定义关系时必须设定这个值。设定这个值时，有些特殊的句法可用。如果不想让 Django 创建逆向关系，把 related_name 设为 '+'，或者以 '+' 结尾。例如，下述代码

确保 `User` 模型没有逆向关系：

```
user = models.ForeignKey(User, related_name='+')
```

`related_query_name`

目标模型使用的反向过滤器名称。如果设定了 `related_name`，默认为它的值；否则，默认为模型的名称。

```
# 声明 ForeignKey 时设定 related_query_name
class Tag(models.Model):
    article = models.ForeignKey(Article, related_name="tags",
      related_query_name="tag")
    name = models.CharField(max_length=255)

# 现在反向过滤器的名称就是它了
Article.objects.filter(tag__name="important")
```

`to_field`

所关联对象上建立关系的字段。Django 默认使用所关联对象的主键。

`db_constraint`

控制是否在数据库中为外键创建约束。默认为 `True`，多数情况下这正是所需的行为。设为 `False` 极有可能破坏数据完整性。尽管如此，有些时候还是需要设为 `False`：

- 有无效的旧数据
- 创建数据库分片

如果设为 `False`，访问不存在的关联对象时抛出 `DoesNotExist` 异常。

`on_delete`

外键引用的对象被删除时，Django 默认模拟 SQL 约束 `ON DELETE CASCADE` 的行为，把包含外键的对象也删除。设定 `on_delete` 参数可以覆盖这个行为。假设外键字段接受 `null` 值，在关联的对象被删除时想把外键设为 `null`，可以这么做：

```
user = models.ForeignKey(User, blank=True, null=True, on_delete=models.SET_NULL)
```

`on_delete` 可取的值在 `django.db.models` 中定义：

- `CASCADE`：层叠删除；默认值。
- `PROTECT`：抛出 `ProtectedError`（`django.db.IntegrityError` 的子类），禁止删除被引用的对象。
- `SET_NULL`：把外键设为 `null`；仅当 `null` 为 `True` 时才能这么做。
- `SET_DEFAULT`：把外键设为默认值；必须为外键设定默认值。

`swappable`

控制外键指向可交换的模型时迁移框架的反应。设为 `True` 时（默认值），如果外键指向的模型与 `settings.AUTH_USER_MODEL`（或其他可交换的模型设置）的当前值匹配，迁移中的关系引用这个设置，而不直接引用模型。

仅当确定模型应该始终指向换入的模型（例如专为自定义的用户模型设计的个人资料模型）时才应该覆盖这个参数，设为 False。设为 False 并不是指换出后可以引用可交换的模型，而是指迁移中的外键指向你指定的模型（比如说，尝试使用不支持的 User 模型时会出大错）。如有迟疑，别去改它，使用默认的 True。

A.4.2 ManyToManyField

多对多关系。有个必须的位置参数：与之关联的类（与 ForeignKey 的那个参数作用完全一样，包括递归关系和惰性关系）。可以通过这个字段的 RelatedManager 添加、删除或创建关联的对象。

数据库表述

为了表示多对多关系，Django 在背后会创建一个中间联结表。联结表的名称默认使用多对多字段的名称和多对多字段所在模型对应的表的名称合成。

有些数据库限制表名的长度，此时表名会被截断，后加一个唯一的哈希值，构成 64 个字符。因此，你可能会看到名为 author_books_9cdf4 的表——这是相当正常的。联结表的名称可以使用 db_table 选项设定。

参数

ManyToManyField 接受一些额外的参数（全为可选的），用于控制关系的功能。

related_name

同 ForeignKey.related_name。

related_query_name

同 ForeignKey.related_query_name。

limit_choices_to

同 ForeignKey.limit_choices_to。如果通过 through 参数自定义了中间联结表，ManyToManyField 的 limit_choices_to 参数没有作用。

symmetrical

仅当与自身建立多对多关系时有用。以下述代码为例：

```
from django.db import models

class Person(models.Model):
    friends = models.ManyToManyField("self")
```

Django 处理这个模型时，发现与自身建立了多对多关系，因此不会为 Person 类添加 person_set 属性，而是假定这是对称的多对多关系，即如果我是你的朋友，你就是我的朋友。

如果不想与自身建立对称的多对多关系，把 symmetrical 设为 False。这样，Django 会为反向关系添加描述符，把多对多关系变成不对称的。

through

Django 会自动生成一个表，用于管理多对多关系。然而，如果想自定义中间表，可以通过 through 选项指定

表示中间表的 Django 模型。

需要在多对多关系中增添额外数据时经常这样做。即便不明确设定 through，仍然有个隐含的模型类，通过它可以直接访问管理关系的表。这个隐含的模型有三个字段：

- id：关系的主键。
- <containing_model>_id：声明 ManyToManyField 那个模型的 id。
- <other_model>_id：ManyToManyField 指向的那个模型的 id。

这个模型可以像常规的模型那样使用，用于查询关联的记录。

throughfields

仅当自定义了中间模型时有用。Django 通常能自行判断使用哪些字段建立多对多关系。

db_table

存储多对多数据的表名。如果不设定，Django 会基于定义关系的模型对应的表名和字段的表名生成一个。

db_constraint

控制是否在数据库中的中间表上为外键创建约束。默认为 True，多数情况下这正是所需的行为；设为 False 极有可能破坏数据完整性。

尽管如此，有些时候还是需要设为 False：

- 有无效的旧数据
- 创建数据库分片

不能同时设定 db_constraint 和 through。

swappable

控制多对多字段指向可交换的模型时迁移框架的反应。设为 True 时（默认值），如果多对多字段指向的模型与 settings.AUTH_USER_MODEL（或其他可交换的模型设置）的当前值匹配，迁移中的关系引用这个设置，而不直接引用模型。

仅当确定模型应该始终指向换入的模型（例如专为自定义的用户模型设计的个人资料模型）时才应该覆盖这个参数，设为 False。如有迟疑，别去改它，使用默认的 True。多对多字段不支持 validators。null 没有效果，因为这样在数据库层无法引用关系。

A.4.3 OneToOneField

一对一关系。理论上，这与设定了 unique=True 的外键字段一样，但是关系的另一端只返回一个对象。一个模型的主键以某种方式扩展另一个模型时最常建立这种关系。多表继承的实现方式是，在子模型和父模型之间建立隐含的一对一关系。

有个必须的位置参数：与之关联的类。这与 ForeignKey 的那个参数作用完全一样，参数也一样（不管是递归关系还是惰性关系）。如果不为 OneToOneField 指定 related_name 参数，Django 默认使用当前模型名称的小写形式。以下述代码为例：

```
from django.conf import settings
from django.db import models

class MySpecialUser(models.Model):
    user = models.OneToOneField(settings.AUTH_USER_MODEL)
    supervisor = models.OneToOneField(settings.AUTH_USER_MODEL,
      related_name='supervisor_of')
```

得到的 User 模型具有下述属性：

```
>>> user = User.objects.get(pk=1)
>>> hasattr(user, 'myspecialuser')
True
>>> hasattr(user, 'supervisor_of')
True
```

如果关联的表中没有关联的记录，访问反向关系时会抛出 DoesNotExist 异常。假如某个用户没有通过 MySpecialUser 指派的主管：

```
>>> user.supervisor_of
Traceback (most recent call last):
    ...
DoesNotExist: User matching query does not exist.
```

OneToOneField 接受的参数与 ForeignKey 完全一样，此外还有一个参数。

parent_link

设为 True，并且在继承自另一个具体模型的模型中使用时，指明这个字段用于链接回到父类，而不是通常由子类化创建的额外的 OneToOneField。使用示例参见 B.9.3 节。

A.5 模型元数据选项

表 A-3 列出可以在模型内部的 class Meta 中使用的全部模型元数据选项。各选项的详细说明和示例参阅 Django 文档。

表 A-3：模型元数据选项

选项	说明
abstract	设为 True 时表明模型是抽象基类。
app_label	如果定义模型的应用不在 INSTALLED_APPS 中，必须指定所属的应用。
db_table	模型使用的数据库表名称。
db_tablespace	模型使用的数据库表空间。默认为项目的 DEFAULT_TABLESPACE 设置（如果设定了）。如果数据库后端不支持表空间，忽略这个选项。
default_related_name	关联的对象回指这个模型默认使用的名称。默认为 <model_name>_set。
get_latest_by	模型中可排序字段的名称，通常是一个 DateField、DateTimeField 或 IntegerField。

（续）

选项	说明
managed	默认为 True，即让 Django 在迁移中创建适当的数据库表，并在执行 flush 管理命令时把表删除。
order_with_respect_to	标记对象为可排序的，排序依据是指定的字段。
ordering	对象的默认排序，获取对象列表时使用。
permissions	创建对象时写入权限表的额外权限。
default_permissions	默认为 ('add', 'change', 'delete')。
proxy	设为 True 时，定义为另一个模型的子类的模型视作代理模型。
select_on_save	指明是否让 Django 使用 1.6 版之前的 django.db.models.Model.save() 算法。
unique_together	设定组合在一起时必须唯一的多个字段名称。
index_together	设定在一起建立索引的多个字段名称。
verbose_name	为对象设定人类可读的名称（单数）。
verbose_name_plural	设定对象的复数名称。

附录 B 数据库 API 参考指南

Django 的数据库 API 是附录 A 所述模型 API 的另一半。定义好模型之后，便可以使用这个 API 访问数据库。本书举了很多使用这个 API 的例子，这一篇附录详述各个选项。

本篇附录以下述博客应用程序中的模型为例：

```
from django.db import models

class Blog(models.Model):
    name = models.CharField(max_length=100)
    tagline = models.TextField()

    def __str__(self):
        return self.name

class Author(models.Model):
    name = models.CharField(max_length=50)
    email = models.EmailField()

    def __str__(self):
        return self.name

class Entry(models.Model):
    blog = models.ForeignKey(Blog)
    headline = models.CharField(max_length=255)
    body_text = models.TextField()
    pub_date = models.DateField()
    mod_date = models.DateField()
    authors = models.ManyToManyField(Author)
    n_comments = models.IntegerField()
    n_pingbacks = models.IntegerField()
    rating = models.IntegerField()

    def __str__(self):
        return self.headline
```

B.1 创建对象

Django 采用一种直观的方式以 Python 对象表述数据库表中的数据：一个模型类表示一个数据库表，而模型类的实例表示表中的具体记录。

创建对象的方法是把关键字参数传给模型类，然后调用 `save()` 方法将其存入数据库。

假设模型保存在 `mysite/blog/models.py` 文件中，下面举个例子：

```
>>> from blog.models import Blog
```

```
>>> b = Blog(name='Beatles Blog', tagline='All the latest Beatles news.')
>>> b.save()
```

在背后，这段代码执行一个 SQL INSERT 语句。除非明确调用 save() 方法，否则 Django 不会触及数据库。

save() 方法没有返回值。

如果想在一步中创建并保存对象，使用 create() 方法。

B.2 保存对象的变动

若想保存数据库中现有对象的变动，使用 save() 方法。

假设 Blog 的一个实例 b5 已经存入数据库，下述示例修改它的名称，然后更新数据库中的记录：

```
>>> b5.name = 'New name'
>>> b5.save()
```

在背后，这段代码执行一个 SQL UPDATE 语句。除非明确调用 save() 方法，否则 Django 不会触及数据库。

B.2.1 保存 ForeignKey 和 ManyToManyField 字段

更新 ForeignKey 字段的方式与常规的字段完全一样，只需把正确类型的对象赋值给字段。假设相应的 Entry 和 Blog 实例已经存入数据库（这样才能将其检索出来），下述示例更新 Entry 实例 entry 的 blog 属性：

```
>>> from blog.models import Entry
>>> entry = Entry.objects.get(pk=1)
>>> cheese_blog = Blog.objects.get(name="Cheddar Talk")
>>> entry.blog = cheese_blog
>>> entry.save()
```

更新 ManyToManyField 字段的方式稍有不同，要在字段上调用 add() 方法，把记录添加到关系中。下述示例把 Author 实例 joe 添加到 entry 对象上：

```
>>> from blog.models import Author
>>> joe = Author.objects.create(name="Joe")
>>> entry.authors.add(joe)
```

如果想一次性为 ManyToManyField 字段添加多个记录，调用 add() 方法时传入多个参数，如下所示：

```
>>> john = Author.objects.create(name="John")
>>> paul = Author.objects.create(name="Paul")
>>> george = Author.objects.create(name="George")
>>> ringo = Author.objects.create(name="Ringo")
>>> entry.authors.add(john, paul, george, ringo)
```

如果赋值或添加的对象类型不对，Django 会报错。

B.3 检索对象

若想从数据库中检索对象，通过模型类的 Manager 实例构建一个 QuerySet。

QuerySet 实例表示数据库中对象的集合。在 QuerySet 上可调用零个、一个或多个过滤器（filter）。过滤器根

据指定参数缩窄查询结果。用 SQL 术语来说，QuerySet 相当于 SELECT 语句，而过滤器相当于限制子句，如 WHERE 或 LIMIT。

QuerySet 通过模型的 Manager 得到。一个模型至少有一个 Manager，默认名为 objects。Manager 可以直接通过模型类访问，例如：

```
>>> Blog.objects
<django.db.models.manager.Manager object at ...>
>>> b = Blog(name='Foo', tagline='Bar')
>>> b.objects
Traceback:
    ...
AttributeError: "Manager isn't accessible via Blog instances."
```

B.3.1 检索全部对象

从表中检索对象最简单的方式是获取全部对象。为此，在 Manager 上调用 all() 方法：

```
>>> all_entries = Entry.objects.all()
```

all() 方法返回一个 QuerySet 对象，包含数据库表中的全部对象。

B.3.2 使用过滤器检索特定对象

all() 方法返回的 QuerySet 对象包含数据库表中的全部对象。然而，通常只需选取其中部分对象。

为此，要添加过滤条件，精选得到的 QuerySet。最常用的精选方法是：

- filter(**kwargs)：返回一个新的 QuerySet 对象，包含匹配指定查找参数的对象。
- exclude(**kwargs)：返回一个新的 QuerySet 对象，包含不匹配指定查找参数的对象。

查找参数（上述函数签名中的 **kwargs）应该满足 B.3.6 节所述的格式。

串联过滤器

精选 QuerySet 后得到的还是 QuerySet，因此可以通过串联，进一步精选。例如：

```
>>> Entry.objects.filter(
...     headline__startswith='What'
... ).exclude(
...     pub_date__gte=datetime.date.today()
... ).filter(pub_date__gte=datetime(2005, 1, 30)
... )
```

上述代码先获取包含数据库表中全部对象的 QuerySet，然后添加一个过滤器，排除一些对象，最后再添加一个过滤器。最终得到的 QuerySet 包含标题以“What”开头、在 2005 年 1 月 30 日至今天之间发布的文章。

过滤后的 QuerySet 是独一无二的

每次精选 QuerySet 得到的都是全新的 QuerySet，与之前的 QuerySet 没有任何联系。精选后得到的 QuerySet 是独立的、截然不同的，可以存储、使用和复用。

例如：

```
>>> q1 = Entry.objects.filter(headline__startswith="What")
>>> q2 = q1.exclude(pub_date__gte=datetime.date.today())
>>> q3 = q1.filter(pub_date__gte=datetime.date.today())
```

这三个 QuerySet 是相互独立的。第一个 QuerySet 包含所有标题以“What”开头的文章；第二个 QuerySet 是第一个的子集，多了一个条件，即排除 pub_date 为今天或未来某天的记录；第三个 QuerySet 是第一个的子集，多了一个条件，即只选择 pub_date 为今天或未来某天的记录。在后续精选的过程中，第一个 QuerySet（q1）不受影响。

QuerySet 是惰性的

QuerySet 是惰性的，创建 QuerySet 不涉及任何数据库活动。你可以一直串联过滤器，Django 并不会执行查询；真正的查询等到求值 QuerySet 时才执行。下面举个例子：

```
>>> q = Entry.objects.filter(headline__startswith="What")
>>> q = q.filter(pub_date__lte=datetime.date.today())
>>> q = q.exclude(body_text__icontains="food")
>>> print(q)
```

看起来这段代码好像访问了三次数据库，但其实只访问了一次——执行最后一行时（print(q)）。通常，直到查看 QuerySet 的结果时才会从数据库中获取数据。届时，求值 QuerySet，访问数据库。

B.3.3 使用 get() 方法检索单个对象

filter() 方法总是返回一个 QuerySet，即便只有一个对象匹配查询也是如此（只有一个元素的 QuerySet）。

如果知道只有一个对象匹配查询，可以在 Manager 上调用 get() 方法，直接返回那个对象：

```
>>> one_entry = Entry.objects.get(pk=1)
```

与 filter() 方法一样，get() 方法的参数可以指定任何查询表达式，详情参见 B.3.6 节。

注意，get() 和 filter() 对切片 [0] 的处理方式有所不同。如果没有匹配查询的结果，get() 抛出 DoesNotExist 异常。这个异常是执行查询那个模型类的属性；因此，在上述代码中，如果没有主键为 1 的 Entry 对象，Django 抛出 Entry.DoesNotExist 异常。

类似地，如果 get() 返回多个对象，Django 也会报错。此时，抛出 MultipleObjectsReturned 异常（也是模型类的属性）。

B.3.4 其他 QuerySet 方法

从数据库中检索对象最常使用的方法是 all()、get()、filter() 和 exclude()。然而，可用的方法远非这几个。全部 QuerySet 方法参见 QuerySet API 参考指南。

B.3.5 限制 QuerySet 的大小

若想限制结果的数量，使用 Python 的数组切片句法。这等效于 SQL 的 LIMIT 和 OFFSET 子句。

例如，返回前 5 个对象（LIMIT 5）：

```
>>> Entry.objects.all()[:5]
```

下述代码返回第 6 个到第 10 个对象（OFFSET 5 LIMIT 5）：

```
>>> Entry.objects.all()[5:10]
```

不支持负数索引（如 Entry.objects.all()[-1]）。

通常，切割 QuerySet 得到一个新的 QuerySet，而且不求值查询。使用 Python 的步进切片句法时除外。例如，下述代码会执行查询，因为要在前 10 个对象中间隔一个跳出一个对象，构成列表：

```
>>> Entry.objects.all()[:10:2]
```

若想检索单个对象（如 SELECT foo FROM bar LIMIT 1），而非一个列表，别用切片，要用单个索引。

例如，下述代码返回按标题字母顺序排列后的第一个 Entry 对象：

```
>>> Entry.objects.order_by('headline')[0]
```

这基本上等价于：

```
>>> Entry.objects.order_by('headline')[0:1].get()
```

然而，要注意，如果没有匹配指定条件的对象，前者抛出 IndexError，而后者抛出 DoesNotExist。详情参见 get() 方法。

B.3.6 按字段查找

按字段查找相当于在 SQL 中添加 WHERE 子句，方法是在 filter()、exclude() 和 get() 方法中指定关键字参数。查找关键字参数的基本句法是 field__lookuptype=value（中间是两个下划线）。例如：

```
>>> Entry.objects.filter(pub_date__lte='2006-01-01')
```

（基本上）相当于下述 SQL：

```
SELECT * FROM blog_entry WHERE pub_date <= '2006-01-01';
```

查找参数中指定的字段必须是模型字段的名称。不过有个例外：按 ForeignKey 字段查找时，字段的名称后面要添加 _id。此时，参数的值应该是外联模型主键的原始值。例如：

```
>>> Entry.objects.filter(blog_id=4)
```

传入无效的关键字参数时，查找函数抛出 TypeError。

按字段查找可用的条件如下：

- exact
- iexact
- contains
- icontains
- in
- gt
- gte
- lt
- lte
- startswith

- istartswith
- endswith
- iendswith
- range
- year
- month
- day
- week_day
- hour
- minute
- second
- isnull
- search
- regex
- iregex

详细说明和示例参见文档。

B.3.7 跨关系查找

Django 提供了强大而直观的关系查找方式，在背后会自动执行 SQL JOIN 查询。若想跨关系，只需使用两个下划线连接各模型中的字段名称，直到得到所需的字段为止。

下述示例在 name 为 'Beatles Blog' 的 Blog 中检索所有 Entry 对象：

```
>>> Entry.objects.filter(blog__name='Beatles Blog')
```

关系的跨度范围不限。

反过来跨也可以。引用反向关系的方法是使用小写的模型名称。

下述示例检索至少有一个 Entry 对象的 headline 包含 'Lennon' 的 Blog 对象：

```
>>> Blog.objects.filter(entry__headline__contains='Lennon')
```

通过多个关系过滤时，如果某个中间模型没有匹配过滤条件的值，Django 假设有一个空的对象（无效，所有值都为 NULL）。也就是说，此时不会抛出异常。以下述过滤器为例：

```
Blog.objects.filter(entry__authors__name='Lennon')
```

（假设有个关联的 Author 模型）如果没有与文章关联的 author 对象，Django 假设也没有相应的 name，以防因缺少 author 对象而抛出异常。通常，这正是所需的行为。然而，使用 isnull 时可能会让人困惑。因此：

```
Blog.objects.filter(entry__authors__name__isnull=True)
```

将返回 author 的 name 值为空的 Blog 对象，以及 entry 的 author 值为空的对象。如果不想要后面那部分对象，代码可以这样写：

```
Blog.objects.filter(entry__authors__isnull=False, entry__authors__name__isnull=True)
```

跨多值关系

基于 ManyToManyField 或反向 ForeignKey 过滤时，可能会用到两种过滤器。以 Blog 和 Entry 之间的（一对多）关系为例。我们可能想找出标题中带有“Lennon”，而且发布于 2008 年的文章所属的博客。

或者，找出标题中带有“Lennon”的文章所属的博客，同时再找出发布于 2008 年的文章所属的博客。因为一个博客关联着多篇文章，所以在某些情况下可能需要做这两种查询。

多对多关系中也有这种需求。假如 Entry 模型有个 ManyToManyField，名为 tags，我们可能想分别找出与“music”和“bands”关联的文章，或者找出标签为“music”，而且状态为“public”的文章。

对这些情况来说，Django 处理 filter() 和 exclude() 的方式是一致的。filter() 调用中的参数在过滤时同时匹配，找出满足全部条件的对象。

多次调用 filter() 用于进一步筛选对象，但是对多值关系来说，过滤的是任何与主模型链接的对象，而不必一定是前一个 filter() 调用选出的对象。

这样说可能有点难以理解，希望通过示例能让你弄明白。为了找出标题中带有“Lennon”，而且发布于 2008 年的文章（同时满足两个条件的文章）所属的博客，可以这样写：

```
Blog.objects.filter(entry__headline__contains='Lennon',
        entry__pub_date__year=2008)
```

为了找出标题中带有“Lennon”的文章所属的博客，同时再找出发布于 2008 年的文章所属的博客，可以这样写：

```
Blog.objects.filter(entry__headline__contains='Lennon').filter(
        entry__pub_date__year=2008)
```

假设只有一个博客中有标题中包含“Lennon”的文章和发布于 2008 年的文章，但是 2008 年发布的文章，标题中都没有“Lennon”。那么，第一个查询不返回任何博客，而第二个查询返回那个博客。

在第二个示例中，第一个过滤器限制查询集合，只包含标题中带有“Lennon”的文章所属的博客；第二个过滤器进一步限制查询集合，限制博客中还要有发布于 2008 年的文章。

第二个过滤器选择的文章可能与第一个过滤器选择的相同，也可能不同。每个过滤器过滤的都是 Blog 对象集合，而非 Entry 对象集合。

这些行为也适用于 exclude()：一个 exclude() 调用中的所有条件同时应用到一个实例上（前提是条件针对的是相同的多值关系）。连续调用 filter() 或 exclude() 处理相同的关系时，后面的调用过滤的可能是不同的关联对象集合。

B.3.8 过滤器可以引用模型中的字段

目前所举的例子在过滤器中都是比较模型字段的值和常量。那么，如果想比较同一个模型中的两个字段的值呢？

为此，Django 提供了 F 表达式。在查询中，F() 的实例引用一个模型字段。在查询过滤器中可以使用这样的引用，以便比较同一个模型实例中的两个字段。

例如，为了找出评论数量比 pingback 多的博客文章，可以构建一个 F() 对象，引用 pingback 数量，然后在查询中使用那个 F() 对象：

```
>>> from django.db.models import F
>>> Entry.objects.filter(n_comments__gt=F('n_pingbacks'))
```

Django 支持 F() 对象与常量和其他 F() 对象做加、减、乘、除、模和幂运算。为了找出评论数量是 pingback 两倍多的博客文章，可以这样修改查询：

```
>>> Entry.objects.filter(n_comments__gt=F('n_pingbacks')* 2)
```

若想找出得分低于 pingback 和评论数量之和的文章，可以这样查询：

```
>>> Entry.objects.filter(rating__lt=F('n_comments') + F('n_pingbacks'))
```

此外，还可以在 F() 对象中使用双下划线表示法跨关系。此时，为了访问关联的对象，会引入适当的联结查询。

例如，若想检索作者姓名与博客名称一样的文章，可以这样查询：

```
>>> Entry.objects.filter(authors__name=F('blog__name'))
```

遇到日期和日期时间字段，可以加上或减去一个 timedelta 对象。下述查询返回发布三天后修改过的文章：

```
>>> from datetime import timedelta
>>> Entry.objects.filter(mod_date__gt=F('pub_date') + timedelta(days=3))
```

通过 .bitand() 和 .bitor()，还可以让 F() 对象支持位运算，例如：

```
>>> F('somefield').bitand(16)
```

B.3.9 pk 简记法

为了方便，Django 为主键提供了简记法——pk。

在 Blog 示例模型中，主键是 id 字段，因此下述三个语句是等效的：

```
>>> Blog.objects.get(id__exact=14) # 明确指定
>>> Blog.objects.get(id=14)        # 暗含 __exact
>>> Blog.objects.get(pk=14)        # pk 暗指 id__exact
```

pk 不限于只能在 __exact 查询中使用，任何查询词条都可以与 pk 联合起来，在模型的主键上执行查询：

```
# 获取 ID 为 1、4 和 7 的博客文章
>>> Blog.objects.filter(pk__in=[1,4,7])

# 获取 ID > 14 的博客文章
>>> Blog.objects.filter(pk__gt=14)
```

在联结查询中也可以使用 pk。例如，下述三个语句是等效的：

```
>>> Entry.objects.filter(blog__id__exact=3) # 明确指定
>>> Entry.objects.filter(blog__id=3)        # 暗含 __exact
>>> Entry.objects.filter(blog__pk=3)        # __pk 暗指 __id__exact
```

B.3.10 转义 LIKE 语句中的百分号和下划线

按字段查找时，如果得到的 SQL 包含 LIKE 语句（iexact、contains、icontains、startswith、istartswith、endswith 和 iendswith），LIKE 语句中的两个特殊字符会自动转义：百分号和下划线。（在 LIKE 语句中，百分号是匹配多个字符的通配符，而下划线是匹配单个字符的通配符。）

这样，查询才能按预期正确运行，不出意外。例如，若想检索标题中包含百分号的文章，像正常字符那样使用百分号即可：

```
>>> Entry.objects.filter(headline__contains='%')
```

转义由 Django 负责；得到的 SQL 类似下面这样：

```
SELECT ... WHERE headline LIKE '%\%%';
```

下划线同理。百分号和下划线都由 Django 默默处理。

B.3.11 查询集合缓存

为了减少数据库访问，每个 QuerySet 都包含缓存。为了写出最高效的代码，一定要理解缓存的工作原理。

对新建的 QuerySet 来说，缓存是空的。首次求值 QuerySet 时（因此访问了数据库），Django 把查询结果保存在 QuerySet 的缓存中，然后返回明确请求的对象（例如，迭代 QuerySet 时返回下一个元素）。后续再求值 QuerySet，则复用缓存的结果。

记住有这么一个缓存，如果对 QuerySet 处理不当，你会深受其害。例如，下述两个语句会创建两个 QuerySet，求值后就丢弃：

```
>>> print([e.headline for e in Entry.objects.all()])
>>> print([e.pub_date for e in Entry.objects.all()])
```

也就是说，相同的数据库查询会执行两次，因此数据库负载也加倍了。此外，两次得到的结果也可能不同，因为在两次查询的间隙可能增加或删除了 Entry 对象。

为了避免这种问题，只需保存 QuerySet，然后复用：

```
>>> queryset = Entry.objects.all()
>>> print([p.headline for p in queryset]) # 求值查询结合
>>> print([p.pub_date for p in queryset]) # 复用缓存中的求值结果
```

查询集合何时不缓存

查询集合并非始终缓存结果。求值查询集合的一部分时会检查缓存，如果未填充，后续查询返回的结果不会缓存。具体而言，通过数组切片或索引限制查询集合的大小时，不会填充缓存。

例如，在查询集合上多次获取某个索引时，每一次都会查询数据库：

```
>>> queryset = Entry.objects.all()
>>> print queryset[5] # 查询数据库
>>> print queryset[5] # 再次查询数据库
```

然而，如果整个查询集合已经求值，则会使用缓存：

```
>>> queryset = Entry.objects.all()
>>> [entry for entry in queryset] # 查询数据库
>>> print queryset[5]             # 使用缓存
>>> print queryset[5]             # 使用缓存
```

下面举例说明其他会导致整个查询集合求值的操作（进而填充缓存）：

```
>>> [entry for entry in queryset]
```

```
>>> bool(queryset)
>>> entry in queryset
>>> list(queryset)
```

B.4 使用 Q 对象执行复杂的查找

filter() 等的关键字参数指定的条件是并联在一起的，如果需要执行更复杂的查询（例如，使用 OR 语句查询），可以使用 Q 对象。

Q 对象（django.db.models.Q）用于封装一系列关键字参数。指定这些关键字参数的方式与前述按字段查询一样。

例如，下述 Q 对象封装一个 LIKE 查询：

```
from django.db.models import Q
Q(question__startswith='What')
```

Q 对象可以使用 & 和 | 运算符结合在一起。此时，两个 Q 对象产出一个新 Q 对象。

例如，下述语句产出一个 Q 对象，通过 OR 把两个 "question__startswith" 查询合并到一起：

```
Q(question__startswith='Who') | Q(question__startswith='What')
```

等效的 SQL WHERE 子句如下：

```
WHERE question LIKE 'Who%' OR question LIKE 'What%'
```

使用 & 和 | 运算符，再利用括号分组，可以组合出任意复杂的查询。此外，Q 对象还可以使用 ~ 运算符取反。因此，可以把常规的查询和取反（NOT）查询合并到一起：

```
Q(question__startswith='Who') | ~Q(pub_date__year=2005)
```

接受关键字参数的查找函数（如 filter()、exclude()、get()）都可以通过位置（无名）参数传入一个或多个 Q 对象。如果把多个 Q 对象传给查找函数，各个 Q 对象并联在一起。例如：

```
Poll.objects.get(
    Q(question__startswith='Who'),
    Q(pub_date=date(2005, 5, 2)) | Q(pub_date=date(2005, 5, 6))
)
```

这段代码基本上相当于下述 SQL：

```
SELECT * from polls WHERE question LIKE 'Who%' AND (pub_date = '2005-05-02' OR pub_date =
'2005-05-06')
```

查找函数可以混用 Q 对象和关键字参数。传给查找函数的所有参数（关键字参数或 Q 对象）并联在一起。然而，如果有 Q 对象，必须放在关键字参数前面。例如：

```
Poll.objects.get(
    Q(pub_date=date(2005, 5, 2)) | Q(pub_date=date(2005, 5, 6)),
    question__startswith='Who')
```

这是有效的查询，与前例作用相同。但是，下述查询是无效的。

```
# 无效查询
```

```
Poll.objects.get(
    question__startswith='Who',
    Q(pub_date=date(2005, 5, 2)) | Q(pub_date=date(2005, 5, 6)))
```

B.5 比较对象

若想比较两个模型实例，使用标准的 Python 比较运算符（双等号，`==`）即可。在背后比较的是两个模型实例的主键值。

以前面的 `Entry` 模型为例，下述两个语句是等效的：

```
>>> some_entry == other_entry
>>> some_entry.id == other_entry.id
```

模型的主键不是 `id` 也没问题。不管主键的名称是什么，比较的始终是主键。假如模型的主键字段名为 `name`，下述两个语句是等效的：

```
>>> some_obj == other_obj
>>> some_obj.name == other_obj.name
```

B.6 删除对象

为了方便，删除方法名为 `delete()`。这个方法立即删除对象，而且没有返回值。例如：

```
e.delete()
```

对象也可以批量删除。`QuerySet` 都有 `delete()` 方法，其作用是删除 `QuerySet` 中的所有成员。

例如，下述代码删除 `pub_date` 的年份为 2005 的所有 `Entry` 对象：

```
Entry.objects.filter(pub_date__year=2005).delete()
```

注意，只要可能，批量删除操作纯粹使用 SQL 语句删除，因此在此期间不一定会调用各个实例的 `delete()` 方法。如果在模型类中自定义了 `delete()` 方法，想调用它的话，必须逐个删除模型实例（例如，迭代 `QuerySet`，在各个对象上调用 `delete()`），而不能在 `QuerySet` 上调用 `delete()` 方法批量删除。

删除对象时，Django 默认模拟 SQL 约束 `ON DELETE CASCADE` 的行为。也就是说，如果一个对象通过外键指向要删除的对象，它自身也会被删除。例如：

```
b = Blog.objects.get(pk=1)
# 删除博客及其中的所有 `Entry` 对象
b.delete()
```

这种层叠行为可以通过 `ForeignKey` 的 `on_delete` 参数自定义。

注意，`delete()` 是 `QuerySet` 上唯一不通过 `Manager` 开放的方法。这是一种防护机制，以防你不小心调用 `Entry.objects.delete()`，把所有文章都删掉。如果确实想删除所有对象，要明确请求完整的查询集合：

```
Entry.objects.all().delete()
```

B.7 复制模型实例

没有内置的方法能复制模型实例，但是能轻易创建所有字段的值都一样的新实例。最简单的方法是把 `pk` 设为

None。以 Blog 模型为例：

```
>>> blog = Blog(name='My blog', tagline='Blogging is easy')
>>> blog.save() # blog.pk == 1
>>> blog.pk = None
>>> blog.save() # blog.pk == 2
```

如果涉及继承，情况要复杂一些。以 Blog 的子类为例：

```
>>> class ThemeBlog(Blog):
...     theme = models.CharField(max_length=200)

>>> django_blog = ThemeBlog(name='Django', tagline='Django is easy', theme='python')
>>> django_blog.save() # django_blog.pk == 3
```

鉴于继承的运作方式，必须把 pk 和 id 都设为 None：

```
>>> django_blog.pk = None
>>> django_blog.id = None
>>> django_blog.save() # django_blog.pk == 4
```

这个过程不会复制关联的对象。如果想复制关系，要多写一些代码。在下述示例中，Entry 与 Author 之间是多对多关系：

```
>>> entry = Entry.objects.all()[0] # 某篇文章
>>> old_authors = entry.authors.all()
>>> entry.pk = None
>>> entry.save()
>>> entry.authors = old_authors # 保存新的多对多关系
```

B.8 一次更新多个对象

若想把 QuerySet 中所有对象的某个字段设为相同的值，可以使用 update() 方法。例如：

```
# 更新 2007 年发布的文章的标题
Entry.objects.filter(pub_date__year=2007).update(headline='Everything is the same')
```

这个方法只能用于设定非关系字段和 ForeignKey 字段。更新非关系字段时，以常量形式提供新值。更新 ForeignKey 字段时，把新值设为要指向的那个模型实例。例如：

```
>>> b = Blog.objects.get(pk=1)

# 修改每个 `Entry` 对象，让它归属这个博客
>>> Entry.objects.all().update(blog=b)
```

update() 方法立即执行，返回查询匹配的行数（如果某些行已经是要更新的值了，返回的行数与实际更新的行数不同）。

对要更新的 QuerySet 唯有一个限制：只能访问一个数据库表，即模型的主表。可以通过关联的字段过滤，但是只能更新模型主表中的列。例如：

```
>>> b = Blog.objects.get(pk=1)

# 更新这个博客中所有文章的标题
```

```
>>> Entry.objects.select_related().filter(blog=b).update(headline='Everything is the same')
```

注意，update() 方法直接转换成 SQL 语句。直接更新执行的是批量操作。update() 方法不会在模型上调用 save() 方法，也不会发送 pre_save 或 post_save 信号（调用 save() 时会发送），更不会更新设定了 auto_now 选项的字段。如果想保存 QuerySet 中的各个对象，确保在各个模型实例上调用 save() 方法，无需使用特殊的函数处理，执行迭代查询集合，然后调用 save() 方法：

```
for item in my_queryset:
    item.save()
```

update() 调用也可以使用 F 表达式，根据同一个模型中另一个字段的值更新字段。根据当前值递增计数器时就可以这么做。例如，下述代码递增博客中每篇文章的 pingback 数量：

```
>>> Entry.objects.all().update(n_pingbacks=F('n_pingbacks') + 1)
```

然而，与 filter() 和 exclude() 不同，update() 中的 F() 对象不能涉及联结查询，只能引用要更新那个模型实例中的字段。如果尝试通过 F() 对象引入联结查询，将抛出 FieldError：

```
# 抛出 FieldError
>>> Entry.objects.update(headline=F('blog__name'))
```

B.9 关联的对象

包含关系（ForeignKey、OneToOneField 或 ManyToManyField）的模型实例可以通过便利的 API 访问关联的对象。

以开头的模型为例，Entry 对象 e 可以通过访问 blog 属性获取关联的 Blog 对象：e.blog。

（这个功能在背后使用 Python 描述符实现。你无须关注内部细节，我指出来是为了满足你的好奇心。）

Django 还为关系的另一端（被关联的模型到定义关系的模型之间的链接）提供了存取 API。例如，Blog 对象 b 可以通过 entry_set 属性访问所有关联的 Entry 对象：b.entry_set.all()。

本节的所有示例都基于开头的 Blog、Author 和 Entry 模型。

B.9.1 一对多关系

正向

如果模型中有 ForeignKey，它的实例可以通过一个简单的属性访问关联的（外部）对象。例如：

```
>>> e = Entry.objects.get(id=2)
>>> e.blog # 返回关联的 Blog 对象
```

通过外键属性还可以获取并设定关联的对象。与你所想的一样，除非调用 save() 方法，否则外键不会存入数据库。例如：

```
>>> e = Entry.objects.get(id=2)
>>> e.blog = some_blog
>>> e.save()
```

如果外键设定了 null=True（即允许 NULL 值），可以把值设为 None，移除关系。例如：

```
>>> e = Entry.objects.get(id=2)
```

```
>>> e.blog = None
>>> e.save() # "UPDATE blog_entry SET blog_id = NULL ...;"
```

首次访问关联的对象时，正向一对多关系会缓存起来。后续在相同的对象上访问外键都读取缓存。例如：

```
>>> e = Entry.objects.get(id=2)
>>> print(e.blog)  # 访问数据库，检索关联的 Blog 对象
>>> print(e.blog)  # 不访问数据库，使用缓存
```

注意，QuerySet 的 select_related() 方法会事先递归重新填充所有一对多关系的缓存。例如：

```
>>> e = Entry.objects.select_related().get(id=2)
>>> print(e.blog)  # 不访问数据库，使用缓存
>>> print(e.blog)  # 不访问数据库，使用缓存
```

反向

如果一个模型中有外键，外键指向的那个模型能通过一个 Manager 获取外键所在模型的全部实例。默认情况下，这个 Manager 名为 foo_set，其中 foo 是源模型名称的小写形式。这个 Manager 返回 QuerySet，可以像 B.3 节所述那样过滤和处理。

例如：

```
>>> b = Blog.objects.get(id=1)
>>> b.entry_set.all() # 返回与 b 关联的所有 Entry 对象

# b.entry_set 是一个 Manager，返回 QuerySet
>>> b.entry_set.filter(headline__contains='Lennon')
>>> b.entry_set.count()
```

定义 ForeignKey 时可以通过 related_name 参数覆盖 foo_set 的名称。假如把 Entry 模型中的外键字段改成 blog = ForeignKey(Blog, related_name='entries')，那么上述示例代码就变成了：

```
>>> b = Blog.objects.get(id=1)
>>> b.entries.all() # 返回与 b 关联的所有 Entry 对象

# b.entries 是一个 Manager，返回 QuerySet
>>> b.entries.filter(headline__contains='Lennon')
>>> b.entries.count()
```

使用自定义的反向管理器

默认情况下，反向关系使用的 RelatedManager 是那个模型默认管理器的子类。如果想让特定查询使用不同的管理器，可以使用下述句法：

```
from django.db import models

class Entry(models.Model):
    #...
    objects = models.Manager()  # 默认管理器
    entries = EntryManager()    # 自定义管理器

b = Blog.objects.get(id=1)
b.entry_set(manager='entries').all()
```

如果 EntryManager 的 get_queryset() 方法执行了默认的过滤操作，会应用到 all() 调用上。

当然，指定自定义的反向管理器后还可以调用其中定义的方法：

```
b.entry_set(manager='entries').is_published()
```

处理关联对象的其他方法

除了 B.3 节所述的 QuerySet 方法之外，外键的管理器还提供了用于处理关联对象集合的其他方法。各方法的概述如下（详情参见文档）。

- add(obj1, obj2, …)：把指定模型对象添加到关联的对象集合中。
- create(**kwargs)：新建对象，保存之后放入关联的对象集合中。返回新建的对象。
- remove(obj1, obj2, …)：从关联的对象集合中删除指定的模型对象。
- clear()：删除所有关联的对象。
- set(objs)：替换关联的对象集合。

若想一次赋值多个关联的对象，只需把关联的对象放入任何可迭代的对象。可迭代的对象可以包含对象实例，也可以列出主键的值。例如：

```
b = Blog.objects.get(id=1)
b.entry_set = [e1, e2]
```

这里，e1 和 e2 可以是完整的 Entry 实例，也可以是主键的值。

如果 clear() 可用，先把 entry_set 中现有的对象全部删除，然后再添加可迭代对象（上例中是一个列表）中的对象。如果 clear() 方法不可用，在现有对象的基础上添加可迭代对象中的对象。

本节所述的反向操作都直接影响数据库中的数据。添加、新建和删除都立即自动存入数据库。

B.9.2 多对多关系

多对多关系的两端都有相互访问的 API，使用方法与上述反向一对多关系一样。

唯一的区别是属性的命名：定义 ManyToManyField 的模型使用的属性名与字段本身相同，而反向模型使用源模型的小写名称，后加 '_set'（与反向一对多关系一样）。

为了便于理解，下面举个例子：

```
>>> e = Entry.objects.get(id=3)
>>> e.authors.all() # 返回这个 Entry 对象的所有 Author 对象
>>> e.authors.count()
>>> e.authors.filter(name__contains='John')
>>> a = Author.objects.get(id=5)
>>> a.entry_set.all() # 返回这个 Author 对象的所有 Entry 对象
```

与 ForeignKey 一样，ManyToManyField 也可以指定 related_name。在上述示例中，如果 Entry 模型中的 ManyToManyField 指定了 related_name='entries'，那么每个 Author 实例都有 entries 属性，而非 entry_set。

B.9.3 一对一关系

一对一关系与多对一关系很像。如果模型中有 OneToOneField，模型的实例可以通过模型上的一个简单属性访

问关联的对象。

例如：

```
class EntryDetail(models.Model):
    entry = models.OneToOneField(Entry)
    details = models.TextField()

ed = EntryDetail.objects.get(id=2)
ed.entry # 返回关联的 Entry 对象
```

区别在于反向查询。一对一关系中关联的模型也有一个 Manager，但是它表示一个对象，而不是对象集合：

```
e = Entry.objects.get(id=2)
e.entrydetail # 返回关联的 EntryDetail 对象
```

如果没有关联的对象，Django 抛出 DoesNotExist 异常。

为反向关系赋值的方式与赋值正向关系一样：

```
e.entrydetail = ed
```

B.9.4 通过关联的对象查询

涉及关联对象的查询，其规则与按字段查询一样。指定查询条件时，可以使用关联对象实例自身，也可以使用关联对象的主键值。

假如一个 Blog 对象 b 的 id 为 5，下述三个查询是等效的：

```
Entry.objects.filter(blog=b)    # 使用对象实例查询
Entry.objects.filter(blog=b.id) # 使用实例的 id 查询
Entry.objects.filter(blog=5)    # 直接使用 id
```

B.10 回落到原始 SQL

如果发现查询太复杂，Django 的数据库映射器无法处理，可以退一步，自己动手编写 SQL。

最后，值得指出的一点是，Django 的数据库层只是数据库的接口。你可以通过其他工具、编程语言或数据库框架访问数据库，Django 应用程序的数据库并不是 Django 专用的。

附录 C 通用视图参考指南

第 10 章对通用视图做了介绍，但是没涉及很多重要的细节。这篇附录说明各个通用视图，并简述各视图可用的选项。为了能够理解这里的内容，请先阅读第 10 章。在阅读的过程中可能需要翻回那一章，回顾 `Book`、`Publisher` 和 `Author` 模型是如何定义的。这篇附录中的示例会用到那三个模型。如果想继续深入了解通用视图的高级话题〔例如在基于类的视图中使用混入（mixin）〕，请访问 Django Project 网站。

C.1 通用视图的通用参数

多数通用视图接受大量参数，用于修改视图的行为。其中部分参数在不同视图中的作用是一样的。表 C-1 说明各个通用参数。不管用在哪个通用视图中，这些参数的作用与下表所述的一样。

表 C-1：通用视图的通用参数

参数	说明
`allow_empty`	布尔值，指明没有对象时是否显示页面。设为 `False` 时，如果没有对象可用，视图抛出 404 错误，而不显示空页面。默认为 `True`。
`context_processors`	应用到视图模板上额外的模板上下文处理器列表（除默认的处理器之外）。参见第 8 章。
`extra_context`	一个字典，包含添加到模板上下文中的值。默认为一个空字典。如果字典中的值是可调用对象，通用视图在渲染模板前调用它。
`mimetype`	指定所得文档的 MIME 类型。默认为 `DEFAULT_MIME_TYPE` 设置的值，如果没修改的话，是 `text/html`。
`queryset`	从中读取对象的一个 `QuerySet`（如 `Author.objects.all()`）。`QuerySet` 的说明参见附录 B。多数通用视图需要这个参数。
`template_loader`	用于加载模板的模板加载器。默认为 `django.template.loader`。参见第 8 章。
`template_name`	渲染页面所用模板的完整名称。可用于覆盖从 `QuerySet` 中推导出的默认模板名称。
`template_object_name`	在模板上下文中使用的模板变量的名称。默认为 `'object'`。列出多个对象的视图（如 `object_list` 视图和各个基于日期的视图）将在这个参数值的后面加上 `'_list'`。

C.2 简单的通用视图

`django.views.generic.base` 模块中有几个简单的视图，针对几个常见的使用场景：渲染不需要视图逻辑的模板和重定向。

C.2.1 渲染模板——TemplateView

这个视图把从 URL 中捕获的关键字参数通过上下文传给模板，渲染指定的模板。

以下述 URL 配置为例：

```
from django.conf.urls import url
from myapp.views import HomePageView

urlpatterns = [
    url(r'^$', HomePageView.as_view(), name='home'),
]
```

对于下面这个简单的 views.py 来说，渲染的是 home.html 模板，返回一个列表，列出前 5 篇文章：

```
from django.views.generic.base import TemplateView
from articles.models import Article

class HomePageView(TemplateView):

    template_name = "home.html"

    def get_context_data(self, **kwargs):
        context = super(HomePageView, self).get_context_data(**kwargs)
        context['latest_articles'] = Article.objects.all()[:5]
        return context # 请求 `/`
```

C.2.2 重定向到其他 URL

django.views.generic.base.RedirectView() 重定向到指定 URL。

指定的 URL 中可能包含字典形式的格式字符串，这些格式字符串会使用从 URL 中捕获的参数代换掉。这一步始终执行（即使没有参数传入），因此 URL 中的 % 必须写成 %%，这样在输出时 Python 才会将其转换成一个百分号。

如果指定的 URL 是 None，Django 返回 HttpResponseGone（410）。

示例 views.py：

```
from django.shortcuts import get_object_or_404
from django.views.generic.base import RedirectView

from articles.models import Article

class ArticleCounterRedirectView(RedirectView):

    permanent = False
    query_string = True
    pattern_name = 'article-detail'

    def get_redirect_url(self, *args, **kwargs):
        article = get_object_or_404(Article, pk=kwargs['pk'])
        article.update_counter()
        return super(ArticleCounterRedirectView,
```

```
                        self).get_redirect_url(*args, **kwargs)
```

示例 urls.py：

```
from django.conf.urls import url
from django.views.generic.base import RedirectView

from article.views import ArticleCounterRedirectView, ArticleDetail

urlpatterns = [
    url(r'^counter/(?P<pk>[0-9]+)/$',
        ArticleCounterRedirectView.as_view(),
        name='article-counter'),
    url(r'^details/(?P<pk>[0-9]+)/$',
        ArticleDetail.as_view(),
        name='article-detail'),
    url(r'^go-to-django/$',
        RedirectView.as_view(url='http://djangoproject.com'),
        name='go-to-django'),
]
```

参数

url

重定向的目标 URL，一个字符串。为 None 时抛出 HTTP 410 错误。

pattern_name

重定向的模板 URL 模式名称。反向解析的方式与在视图中传入 *args 和 **kwargs 时一样。

permanent

指定是否永久重定向。唯一的区别是返回的 HTTP 状态码。设为 True 时，重定向使用状态码 301；设为 False 时，重定向使用状态码 302。默认为 True。

query_string

是否把 GET 查询字符串传给新 URL。设为 True 时，查询字符串追加到 URL 末尾；设为 False 时，丢掉查询字符串。默认为 False。

方法

get_redirect_url(*args, **kwargs) 用于构建重定向的目标 URL。

默认的实现以 url 为基础，然后代换 % 标记的具名参数，变成 URL 中具名分组捕获的字符串。

如果未传入 url，get_redirect_url() 尝试使用从 URL 中捕获的信息（具名分组和匿名分组都用上）反向解析 pattern_name。

如果把 query_string 设为 True，还会在生成的 URL 末尾加上查询字符串。

子类可以根据需要实现所需的行为，只要返回一个可以重定向的 URL 字符串即可。

C.3 列表/详情通用视图

列表/详情通用视图针对的常见使用场景是，在一个视图中列出一些元素，然后在单独的视图中显示各元素的详情。

C.3.1 列出对象

django.views.generic.list.ListView

这个视图显示的页面中列出一系列对象。

示例 views.py：

```
from django.views.generic.list import ListView
from django.utils import timezone

from articles.models import Article

class ArticleListView(ListView):

    model = Article

    def get_context_data(self, **kwargs):
        context = super(ArticleListView, self).get_context_data(**kwargs)
        context['now'] = timezone.now()
        return context
```

示例 myapp/urls.py：

```
from django.conf.urls import url

from article.views import ArticleListView

urlpatterns = [
    url(r'^$', ArticleListView.as_view(), name='article-list'),
]
```

示例 myapp/article_list.html：

```
<h1>Articles</h1>
<ul>
{% for article in object_list %}
    <li>{{ article.pub_date|date }} - {{ article.headline }}</li>
{% empty %}
    <li>No articles yet.</li>
{% endfor %}
</ul>
```

C.3.2 详情视图

django.views.generic.detail.DetailView

这个视图显示单个对象的详情。

示例 myapp/views.py：

```python
from django.views.generic.detail import DetailView
from django.utils import timezone

from articles.models import Article

class ArticleDetailView(DetailView):

    model = Article

    def get_context_data(self, **kwargs):
        context = super(ArticleDetailView,
                        self).get_context_data(**kwargs)
        context['now'] = timezone.now()
        return context
```

示例 myapp/urls.py：

```python
from django.conf.urls import url

from article.views import ArticleDetailView

urlpatterns = [
    url(r'^(?P<slug>[-_\w]+)/$',
        ArticleDetailView.as_view(),
        name='article-detail'),
]
```

示例 myapp/article_detail.html：

```html
<h1>{{ object.headline }}</h1>
<p>{{ object.content }}</p>
<p>Reporter: {{ object.reporter }}</p>
<p>Published: {{ object.pub_date|date }}</p>
<p>Date: {{ now|date }}</p>
```

C.4 基于日期的通用视图

django.views.generic.dates 模块中基于日期的通用视图用于显示基于日期的层次数据。

C.4.1 ArchiveIndexView

顶层索引页面，显示最新的对象（按日期排序）。如果未把 allow_future 设为 True，未来日期对应的对象不包含在内。

上下文

除了 django.views.generic.list.MultipleObjectMixin（混入 django.views.generic.dates.BaseDateListView 中）提供的上下文之外，还有：

- date_list：一个 DateQuerySet 对象，包含根据 queryset 判断具有对象的所有年份（以 date-

time.datetime 对象表示），倒序排列。

注意

- context_object_name 的默认值为 latest。
- template_name_suffix 的默认值为 _archive。
- 默认提供的 date_list 按年划分，不过可以通过 date_list_period 属性调整为按月或按日划分。子类也是这样。

示例 myapp/urls.py：

```
from django.conf.urls import url
from django.views.generic.dates import ArchiveIndexView

from myapp.models import Article

urlpatterns = [
    url(r'^archive/$',
        ArchiveIndexView.as_view(model=Article, date_field="pub_date"),
        name="article_archive"),
]
```

示例 myapp/article_archive.html：

```
<ul>
    {% for article in latest %}
        <li>{{ article.pub_date }}: {{ article.title }}</li>
    {% endfor %}
</ul>
```

上述模板输出所有文章。

C.4.2 YearArchiveView

按月归档页面，显示指定年份中的所有月份。如果未把 allow_future 设为 True，未来日期对应的对象不包含在内。

上下文

除了 django.views.generic.list.MultipleObjectMixin（混入 django.views.generic.dates.BaseDateListView 中）提供的上下文之外，还有：

- date_list：一个 DateQuerySet 对象，包含根据 queryset 判断具有对象的所有月份（以 datetime.datetime 对象表示），升序排列。
- year：一个 date 对象，表示指定的年。
- next_year：一个 date 对象，表示由 allow_empty 和 allow_future 确定的下一年的第一天。
- previous_year：一个 date 对象，表示由 allow_empty 和 allow_future 确定的前一年的第一天。

注意

- template_name_suffix 的默认值为 _archive_year。

示例 myapp/views.py：

```
from django.views.generic.dates import YearArchiveView

from myapp.models import Article

class ArticleYearArchiveView(YearArchiveView):
    queryset = Article.objects.all()
    date_field = "pub_date"
    make_object_list = True
    allow_future = True
```

示例 myapp/urls.py：

```
from django.conf.urls import url

from myapp.views import ArticleYearArchiveView

urlpatterns = [
    url(r'^(?P<year>[0-9]{4})/$',
        ArticleYearArchiveView.as_view(),
        name="article_year_archive"),
]
```

示例 myapp/article_archive_year.html：

```
<ul>
    {% for date in date_list %}
        <li>{{ date|date }}</li>
    {% endfor %}
</ul>
<div>
    <h1>All Articles for {{ year|date:"Y" }}</h1>
    {% for obj in object_list %}
        <p>
            {{ obj.title }} - {{ obj.pub_date|date:"F j, Y" }}
        </p>
    {% endfor %}
</div>
```

C.4.3 MonthArchiveView

按月归档页面，显示指定月份中的所有对象。如果未把 allow_future 设为 True，未来日期对应的对象不包含在内。

上下文

除了 MultipleObjectMixin（混入 BaseDateListView 中）提供的上下文之外，还有：

- date_list：一个 DateQuerySet 对象，包含根据 queryset 判断的一月之中具有对象的所有日（以 datetime.datetime 对象表示），升序排列。
- month：一个 date 对象，表示指定的月。

- next_month：一个 date 对象，表示由 allow_empty 和 allow_future 确定的下一月的第一天。
- previous_month：一个 date 对象，表示由 allow_empty 和 allow_future 确定的前一月的第一天。

注意

- template_name_suffix 的默认值为 _archive_month。

示例 myapp/views.py：

```
from django.views.generic.dates import MonthArchiveView

from myapp.models import Article

class ArticleMonthArchiveView(MonthArchiveView):
    queryset = Article.objects.all()
    date_field = "pub_date"
    make_object_list = True
    allow_future = True
```

示例 myapp/urls.py：

```
from django.conf.urls import url

from myapp.views import ArticleMonthArchiveView

urlpatterns = [
    # 比如 /2012/aug/
    url(r'^(?P<year>[0-9]{4})/(?P<month>[-\w]+)/$',
        ArticleMonthArchiveView.as_view(),
        name="archive_month"),
    # 比如 /2012/08/
    url(r'^(?P<year>[0-9]{4})/(?P<month>[0-9]+)/$',
        ArticleMonthArchiveView.as_view(month_format='%m'),
        name="archive_month_numeric"),
]
```

示例 myapp/article_archive_month.html：

```
<ul>
    {% for article in object_list %}
        <li>{{ article.pub_date|date:"F j, Y" }}:
            {{ article.title }}
        </li>
    {% endfor %}
</ul>

<p>
    {% if previous_month %}
        Previous Month: {{ previous_month|date:"F Y" }}
    {% endif %}
    {% if next_month %}
        Next Month: {{ next_month|date:"F Y" }}
    {% endif %}
```

```
</p>
```

C.4.4 WeekArchiveView

按周归档页面，显示指定周中的所有对象。如果未把 allow_future 设为 True，未来日期对应的对象不包含在内。

上下文

除了 MultipleObjectMixin（混入 BaseDateListView 中）提供的上下文之外，还有：

- week：一个 date 对象，表示指定周的第一天。
- next_week：一个 date 对象，表示由 allow_empty 和 allow_future 确定的下一周的第一天。
- previous_week：一个 date 对象，表示由 allow_empty 和 allow_future 确定的前一周的第一天。

注意

- template_name_suffix 的默认值为 _archive_week。

示例 myapp/views.py：

```
from django.views.generic.dates import WeekArchiveView

from myapp.models import Article

class ArticleWeekArchiveView(WeekArchiveView):
    queryset = Article.objects.all()
    date_field = "pub_date"
    make_object_list = True
    week_format = "%W"
    allow_future = True
```

示例 myapp/urls.py：

```
from django.conf.urls import url

from myapp.views import ArticleWeekArchiveView

urlpatterns = [
    # 比如 /2012/week/23/
    url(r'^(?P<year>[0-9]{4})/week/(?P<week>[0-9]+)/$',
        ArticleWeekArchiveView.as_view(),
        name="archive_week"),
]
```

示例 myapp/article_archive_week.html：

```
<h1>Week {{ week|date:'W' }}</h1>

<ul>
    {% for article in object_list %}
        <li>{{ article.pub_date|date:"F j, Y" }}: {{
article.title }}</li>
```

```
        {% endfor %}
    </ul>

    <p>
        {% if previous_week %}
            Previous Week: {{ previous_week|date:"F Y" }}
        {% endif %}
        {% if previous_week and next_week %}--{% endif %}
        {% if next_week %}
            Next week: {{ next_week|date:"F Y" }}
        {% endif %}
    </p>
```

这个示例输出了周几。WeekArchiveView 中，week_format 默认采用的格式是 "%U"，这是美国使用的格式，一周从周日开始。"%W" 则使用 ISO 格式，一周从周一开始。"%W" 在 strftime() 和 date 中的作用与此相同。

然而，date 模板过滤器无法输出美国的周制，"%U" 输出的是距 Unix 纪元的秒数。

C.4.5 DayArchiveView

按日归档页面，显示指定一天中的所有对象。如果未把 allow_future 设为 True，不管未来的日期有没有对应的对象，未来的日都抛出 404 错误。

上下文

除了 MultipleObjectMixin（混入 BaseDateListView 中）提供的上下文之外，还有：

- day：一个 date 对象，表示指定的日。
- next_day：一个 date 对象，表示由 allow_empty 和 allow_future 确定的下一天。
- previous_day：一个 date 对象，表示由 allow_empty 和 allow_future 确定的前一天。
- next_month：一个 date 对象，表示由 allow_empty 和 allow_future 确定的下一月的第一天。
- previous_month：一个 date 对象，表示由 allow_empty 和 allow_future 确定的前一月的第一天。

注意

- template_name_suffix 的默认值为 _archive_day。

示例 myapp/views.py：

```
from django.views.generic.dates import DayArchiveView

from myapp.models import Article

class ArticleDayArchiveView(DayArchiveView):
    queryset = Article.objects.all()
    date_field = "pub_date"
    make_object_list = True
    allow_future = True
```

示例 myapp/urls.py：

```
from django.conf.urls import url
```

```
from myapp.views import ArticleDayArchiveView

urlpatterns = [
    # 比如 /2012/nov/10/
    url(r'^(?P<year>[0-9]{4})/(?P<month>[-\w]+)/(?P<day>[0-9]+)/$',
        ArticleDayArchiveView.as_view(),
        name="archive_day"),
]
```

示例 myapp/article_archive_day.html：

```
<h1>{{ day }}</h1>

<ul>
    {% for article in object_list %}
        <li>
        {{ article.pub_date|date:"F j, Y" }}: {{ article.title }}
        </li>
    {% endfor %}
</ul>

<p>
    {% if previous_day %}
        Previous Day: {{ previous_day }}
    {% endif %}
    {% if previous_day and next_day %}--{% endif %}
    {% if next_day %}
        Next Day: {{ next_day }}
    {% endif %}
</p>
```

C.4.6 TodayArchiveView

按日归档页面，显示今天的所有对象。作用与 django.views.generic.dates.DayArchiveView 完全一样，只不过使用的是今天的日期，而不通过 year/month/day 参数指定。

注意

- template_name_suffix 的默认值为 _archive_today。

示例 myapp/views.py：

```
from django.views.generic.dates import TodayArchiveView

from myapp.models import Article

class ArticleTodayArchiveView(TodayArchiveView):
    queryset = Article.objects.all()
    date_field = "pub_date"
    make_object_list = True
    allow_future = True
```

示例 myapp/urls.py：

```
from django.conf.urls import url

from myapp.views import ArticleTodayArchiveView

urlpatterns = [
    url(r'^today/$',
        ArticleTodayArchiveView.as_view(),
        name="archive_today"),
]
```

怎么不给出 TodayArchiveView 的示例模板了？

这个视图默认使用的视图与 DayArchiveView 一样，因此示例同前。如果想使用其他模板，把 template_name 属性设为新模板的名称。

C.4.7 DateDetailView

呈现单个对象的页面。如果未把 allow_future 设为 True，如果对象对应的是未来日期，默认抛出 404 错误。

上下文

- 在 DateDetailView 中与 model 关联的那个对象。

注意

- template_name_suffix 的默认值为 _detail。

示例 myapp/urls.py：

```
from django.conf.urls import url
from django.views.generic.dates import DateDetailView

urlpatterns = [
    url(r'^(?P<year>[0-9]+)/(?P<month>[-\w]+)/(?P<day>[0-9]+)/

      (?P<pk>[0-9]+)/$',
        DateDetailView.as_view(model=Article, date_field="pub_date"),
        name="archive_date_detail"),
]
```

示例 myapp/article_detail.html：

```
<h1>{{ object.title }}</h1>
```

C.5 在基于类的视图中处理表单

处理表单时要考虑 3 种情况：

- 原来的 GET（为空，或预填的表单）
- POST 发送无效的数据（通常再次显示表单，并指出错误）

- POST 发送有效的数据（通常在处理数据之后重定向）

自己处理这些情况往往要重复编写大量样板代码（参见文档）。鉴于此，Django 提供了一系列基于类的通用视图，用于处理表单。

C.5.1 简单的表单

以下面这个简单的联系表单为例：

```
# forms.py

from django import forms

class ContactForm(forms.Form):
    name = forms.CharField()
    message = forms.CharField(widget=forms.Textarea)

    def send_email(self):
        # 发送邮件时使用 self.cleaned_data 字典中的数据
        pass
```

对应的视图可以使用 FormView 构建：

```
# views.py

from myapp.forms import ContactForm
from django.views.generic.edit import FormView

class ContactView(FormView):
    template_name = 'contact.html'
    form_class = ContactForm
    success_url = '/thanks/'

    def form_valid(self, form):
        # 收到有效的表单数据时调用这个方法
        # 应该返回一个 HttpResponse 对象
        form.send_email()
        return super(ContactView, self).form_valid(form)
```

注意：

- FormView 继承 TemplateResponseMixin，因此这里可以使用 template_name。
- form_valid() 的默认实现只是重定向到 success_url。

C.5.2 模型表单

处理模型时使用通用视图非常方便。只要能确定模型类，通用视图就能自动创建一个 ModelForm。确定模型类的方法如下：

- 使用 model 属性指定的模型类。
- 使用 get_object() 返回的对象所属的类。

- 使用 queryset 对应的模型。

模型表单视图提供的 form_valid() 能自动保存模型。如果有其他需求，可以覆盖这个方法。下文有示例。

无需为 CreateView 或 UpdateView 提供 success_url，目标地址可以使用模型对象的 get_absolute_url() 方法获取。

如果想使用自定义的 ModelForm（例如添加额外的数据验证），只需在视图中设定 form_class。

> **提示**
>
> 指定自定义的表单类时，即便 form_class 可能是 ModelForm 的子类，仍要指定模型。

首先，在 Author 类中添加 get_absolute_url()：

```
# models.py

from django.core.urlresolvers import reverse
from django.db import models

class Author(models.Model):
    name = models.CharField(max_length=200)

    def get_absolute_url(self):
        return reverse('author-detail', kwargs={'pk': self.pk})
```

然后就可以把具体的工作交给 CreateView 等视图去做了。注意基于类的通用视图是如何找到的，自己不用写任何逻辑。

```
# views.py

from django.views.generic.edit import CreateView, UpdateView, DeleteView
from django.core.urlresolvers import reverse_lazy
from myapp.models import Author

class AuthorCreate(CreateView):
    model = Author
    fields = ['name']

class AuthorUpdate(UpdateView):
    model = Author
    fields = ['name']

class AuthorDelete(DeleteView):
    model = Author
    success_url = reverse_lazy('author-list')
```

这里要使用 reverse_lazy()，不能使用 reverse()，因为导入文件时 URL 尚未加载。

fields 属性的作用与 ModelForm 内部 Meta 类的 fields 属性一样。正常情况下，这个属性是必须的，否则视图会抛出 ImproperlyConfigured 异常。

如果同时指定 fields 和 form_class 属性，也会抛出 ImproperlyConfigured 异常。

最后，在 URL 配置中使用这些新视图：

```
# urls.py

from django.conf.urls import url
from myapp.views import AuthorCreate, AuthorUpdate, AuthorDelete

urlpatterns = [
    # ...
    url(r'author/add/$', AuthorCreate.as_view(), name='author_add'),
    url(r'author/(?P<pk>[0-9]+)/$', AuthorUpdate.as_view(),
        name='author_update'),
    url(r'author/(?P<pk>[0-9]+)/delete/$', AuthorDelete.as_view(),
        name='author_delete'),
]
```

在这个示例中：

- `CreateView` 和 `UpdateView` 使用 `myapp/author_form.html`
- `DeleteView` 使用 `myapp/author_confirm_delete.html`

如果想让 `CreateView` 和 `UpdateView` 使用不同的模板，在视图类中设定 `template_name` 或 `template_name_suffix`。

C.5.3 模型和 `request.user`

使用 `CreateView` 创建对象时，若想记录是哪个用户创建的，可以使用自定义的 `ModelForm`。首先，在模型中添加外键关系：

```
# models.py

from django.contrib.auth.models import User
from django.db import models

class Author(models.Model):
    name = models.CharField(max_length=200)
    created_by = models.ForeignKey(User)

    # ...
```

在视图中要确保可编辑的字段列表中没有 `created_by`，然后覆盖 `form_valid()`，添加用户：

```
from django.views.generic.edit import CreateView
from myapp.models import Author

class AuthorCreate(CreateView):
    model = Author
    fields = ['name']

    def form_valid(self, form):
        form.instance.created_by = self.request.user
        return super(AuthorCreate, self).form_valid(form)
```

注意，这个视图要使用 login_required() 装饰，或者在 form_valid() 中使用其他方式处理未授权的用户。

C.5.4 Ajax 示例

下面举个简单的例子，说明如何实现既能处理 Ajax 请求，也能处理常规 POST 请求的表单：

```
from django.http import JsonResponse
from django.views.generic.edit import CreateView
from myapp.models import Author

class AjaxableResponseMixin(object):
    def form_invalid(self, form):
        response = super(AjaxableResponseMixin, self).form_invalid(form)
        if self.request.is_ajax():
            return JsonResponse(form.errors, status=400)
        else:
            return response

    def form_valid(self, form):
        # 一定要调用父类的 form_valid() 方法
        # 因为在那里可能做了些处理
        # （对 CreateView 来说，会调用 form.save()）
        response = super(AjaxableResponseMixin, self).form_valid(form)
        if self.request.is_ajax():
            data = {
                'pk': self.object.pk,
            }
            return JsonResponse(data)
        else:
            return response

class AuthorCreate(AjaxableResponseMixin, CreateView):
    model = Author
    fields = ['name']
```

附录 D Django 设置

Django 设置文件包含项目的所有配置。这篇附录说明设置的作用和可用的设置。

D.1 设置文件是什么?

设置文件其实就是一个 Python 模块，里面定义了模块层变量。下面是几个设置示例：

```
ALLOWED_HOSTS = ['www.example.com']
DEBUG = False
DEFAULT_FROM_EMAIL = 'webmaster@example.com'
```

> **提醒**
>
> 如果把 `DEBUG` 设为 `False`，还要正确设置 `ALLOWED_HOSTS`。

鉴于此，设置文件要遵守下述规则：

- 不能有 Python 句法错误。
- 可以使用常规的 Python 句法动态设定。例如：`MY_SETTING = [str(i) for i in range(30)]`。
- 可以从其他设置文件中导入值。

D.1.1 默认设置

如果某个设置是不需要的，无需在 Django 设置文件中设定。每个设置都有合理的默认值。这些默认值在 `django/conf/global_settings.py` 模块中。编译设置时，Django 采用下述算法：

- 从 `global_settings.py` 中加载设置。
- 从专门的设置文件中加载设置，必要时覆盖全局设置。

注意，设置文件无需导入 `global_settings`，这样做就画蛇添足了。

D.1.2 查看改动的设置

查看与默认值不同的设置有简便的方法。`python manage.py diffsettings` 命令显示当前设置文件与 Django 默认设置之间的差异。详情参见 `diffsettings` 命令的文档。

D.2 在 Python 代码中使用设置

若想在 Django 应用中使用设置，导入 `django.conf.settings` 对象。例如：

```
from django.conf import settings
```

```
if settings.DEBUG:
    # 做些事情
```

注意，django.conf.settings 不是模块，而是对象。因此无法导入单个设置：

```
from django.conf.settings import DEBUG  # 不能这么做
```

还要注意，不能在代码中导入 global_settings 或设置文件。django.conf.settings 对默认设置和项目专用的设置做了抽象，以便统一接口。这样做还解耦了使用设置的代码和设置的位置。

D.3 运行时调整设置

别在运行时调整应用程序的设置。例如，别在视图中这样做：

```
from django.conf import settings
settings.DEBUG = True   # 别这么做！
```

只能在设置文件中为设置赋值。

D.4 安全性

设置文件中有敏感信息，例如数据库密码，因此要尽量限制访问。例如，可以修改文件权限，只让自己和运行 Web 服务器的用户读取。在共享主机环境中尤其要注意。

D.5 自己定义设置

Django 并不禁止你为自己的 Django 应用定义设置。自己定义设置时要遵守下述约定：

- 设置名称为全大写
- 不要再次使用已存在的设置

值为序列时，Django 使用元组，而不是列表；但这只是一个约定。

D.6 DJANGO_SETTINGS_MODULE

使用 Django 时要告诉它你想使用哪个设置文件。指定的方法是使用环境变量 DJANGO_SETTINGS_MODULE。这个环境变量的值应该使用 Python 路径句法，例如 mysite.settings。

D.6.1 django-admin 实用命令

可以使用 django-admin 命令一次性设置这个环境变量，或者每次运行时传入设置模块。例如（Unix Bash shell）：

```
export DJANGO_SETTINGS_MODULE=mysite.settings
django-admin runserver
```

或（Windows shell）：

```
set DJANGO_SETTINGS_MODULE=mysite.settings
django-admin runserver
```

手动指定设置模块使用 `--settings` 命令行参数：

```
django-admin runserver --settings=mysite.settings
```

D.6.2 在服务器中（`mod_wsgi`）

在线上服务器环境中要告诉 WSGI 应用程序使用哪个设置文件。此时使用 `os.environ`：

```
import os

os.environ['DJANGO_SETTINGS_MODULE'] = 'mysite.settings'
```

详情，以及 Django WSGI 应用程序的其他共性，参见第 13 章。

D.7 在不设置 `DJANGO_SETTINGS_MODULE` 的情况下使用设置

有时可能想跳过 `DJANGO_SETTINGS_MODULE` 环境变量。例如，单独使用模板系统时，可能不想设置这样一个指向设置模块的环境变量。此时，可以手动配置 Django 的设置。方法如下：

```
django.conf.settings.configure(default_settings,**settings)
```

例如：

```
from django.conf import settings

settings.configure(DEBUG=True, TEMPLATE_DEBUG=True)
```

`configure()` 接受任意多个关键字参数，每个关键字参数表示一个设置和它的值。参数名称都应该使用全大写字母，而且与真正的设置名称一样。如果后面要使用没传给 `configure()` 的设置，Django 将使用设置的默认值。

在大型应用程序中使用框架的部分组件时基本上必须这么配置 Django，其实，也推荐这么做。因此，通过 `settings.configure()` 配置时，Django 不会对进程的环境变量做任何修改（原因参见 `TIME_ZONE` 的文档）。此时，Django 假定你能完全掌控自己的环境。

D.7.1 自定义默认设置

如果想从 `django.conf.global_settings` 之外的模块加载默认值，调用 `configure()` 时可以把 `default_settings` 参数（或第一个位置参数）设为提供默认值的模块或类。在下述示例中，默认值来自 `myapp_defaults`，而且不管 `myapp_defaults` 中 `DEBUG` 为何值，都把 `DEBUG` 设为 `True`：

```
from django.conf import settings
from myapp import myapp_defaults

settings.configure(default_settings=myapp_defaults, DEBUG=True)
```

下述示例通过位置参数设定 `myapp_defaults`，与前例等效：

```
settings.configure(myapp_defaults, DEBUG=True)
```

正常情况下，无需这样覆盖默认值。Django 提供的默认值足够合理，可以放心使用。注意，如果传入了新的默认模块，整个 Django 默认值都将被替换掉，因此必须为代码中可能用到的每个设置指定值。完整设置参见 `django.conf.settings.global_settings` 模块。

D.7.2 configure() 和 DJANGO_SETTINGS_MODULE 必选其一

如果未设定 DJANGO_SETTINGS_MODULE 环境变量，在读取设置之前必须调用 configure()。否则，首次访问设置时，Django 将抛出 ImportError 异常。如果设定了 DJANGO_SETTINGS_MODULE，也访问了设置，而后又调用 configure()，Django 将抛出 RuntimeError 异常，指明设置已经配置。为此，有个专门的特性（property）：

```
django.conf.settings.configured
```

例如：

```
from django.conf import settings
if not settings.configured:
    settings.configure(myapp_defaults, DEBUG=True)
```

此外，不能多次调用 configure()，或者访问设置之后再次调用 configure()。综上，configure() 和 DJANGO_SETTINGS_MODULE 只能二选其一。不能同时使用，也不能都不用。

D.8 可用设置

Django 提供了大量设置。为了便于引用，我把设置分成了六类，一类对应一个表。

1. 核心设置（表 D-1）
2. 身份验证设置（表 D-2）
3. 消息设置（表 D-3）
4. 会话设置（表 D-4）
5. Django 网站设置（表 D-5）
6. 静态文件设置（表 D-6）

每个表中都列出了可用的设置及其默认值。各个设置的详细说明和用法举例参见 Django 文档。

> **提醒**
>
> 覆盖设置时要小心，尤其是覆盖默认值为非空列表或字典的设置，例如 MIDDLEWARE_CLASSES 和 STATICFILES_FINDERS。别把想使用的 Django 功能对应的组件删掉了。

D.8.1 核心设置

表 D-1：Django 核心设置

设置	默认值
ABSOLUTE_URL_OVERRIDES	{}（空字典）
ADMINS	[]（空列表）
ALLOWED_HOSTS	[]（空列表）
APPEND_SLASH	True
CACHE_MIDDLEWARE_ALIAS	default

（续）

设置	默认值

CACHES	{
	    'default': {
	        'BACKEND':
	'django.core.cache.backends.locmem.LocMemCache',
	    }
	}

CACHE_MIDDLEWARE_KEY_PREFIX	''（空字符串）
CACHE_MIDDLEWARE_SECONDS	600
CSRF_COOKIE_AGE	31449600（1 年对应的秒数）
CSRF_COOKIE_DOMAIN	None
CSRF_COOKIE_HTTPONLY	False
CSRF_COOKIE_NAME	'csrftoken'
CSRF_COOKIE_PATH	'/'
CSRF_COOKIE_SECURE	False
DATE_INPUT_FORMATS	('%Y-%m-%d', '%m/%d/%Y', '%m/%d/%y', '%b %d %Y', '%b %d, %Y', '%d %b %Y', '%d %b, %Y', '%B %d %Y', '%B %d, %Y', '%d %B %Y', '%d %B, %Y',)
DATETIME_FORMAT	'N j, Y, P'（如 Feb. 4, 2003, 4 p.m.）
DATETIME_INPUT_FORMATS	('%Y-%m-%d %H:%M:%S', '%Y-%m-%d %H:%M:%S.%f', '%Y-%m-%d %H:%M', '%Y-%m-%d', '%m/%d/%Y %H:%M:%S', '%m/%d/%Y %H:%M:%S.%f', '%m/%d/%Y %H:%M', '%m/%d/%Y', '%m/%d/%y %H:%M:%S', '%m/%d/%y %H:%M:%S.%f', '%m/%d/%y %H:%M', '%m/%d/%y',)
DEBUG	False
DEBUG_PROPAGATE_EXCEPTIONS	False
DECIMAL_SEPARATOR	'.'（点号）

（续）

设置	默认值
DEFAULT_CHARSET	'utf-8'
DEFAULT_CONTENT_TYPE	'text/html'
DEFAULT_EXCEPTION_REPORTER_FILTER	django.views.debug.SafeExceptionReporterFilter
DEFAULT_FILE_STORAGE	django.core.files.storage.FileSystemStorage
DEFAULT_FROM_EMAIL	'webmaster@localhost'
DEFAULT_INDEX_TABLESPACE	''（空字符串）
DEFAULT_TABLESPACE	''（空字符串）
DISALLOWED_USER_AGENTS	[]（空列表）
EMAIL_BACKEND	django.core.mail.backends.smtp.EmailBackend
EMAIL_HOST	'localhost'
EMAIL_HOST_PASSWORD	''（空字符串）
EMAIL_HOST_USER	''（空字符串）
EMAIL_PORT	25
EMAIL_SUBJECT_PREFIX	'[Django] '
EMAIL_USE_TLS	False
EMAIL_USE_SSL	False
EMAIL_SSL_CERTFILE	None
EMAIL_SSL_KEYFILE	None
EMAIL_TIMEOUT	None
FILE_CHARSET	'utf-8'
FILE_UPLOAD_HANDLERS	("django.core.files.uploadhandler.MemoryFileUploadHandler", "django.core.files.uploadhandler.TemporaryFileUploadHandler")
FILE_UPLOAD_MAX_MEMORY_SIZE	2621440（即 2.5 MB）
FILE_UPLOAD_DIRECTORY_PERMISSIONS	None
FILE_UPLOAD_PERMISSIONS	None
FILE_UPLOAD_TEMP_DIR	None
FIRST_DAY_OF_WEEK	0（星期天）
FIXTURE_DIRS	[]（空列表）
FORCE_SCRIPT_NAME	None

设置	默认值
FORMAT_MODULE_PATH	None
IGNORABLE_404_URLS	[]（空列表）
INSTALLED_APPS	[]（空列表）
INTERNAL_IPS	[]（空列表）
LANGUAGE_CODE	'en-us'
LANGUAGE_COOKIE_AGE	None（关闭浏览器后失效）
LANGUAGE_COOKIE_DOMAIN	None
LANGUAGE_COOKIE_NAME	'django_language'
LANGUAGES	全部可用语言构成的列表
LOCALE_PATHS	[]（空列表）
LOGGING	配置日志的字典
LOGGING_CONFIG	'logging.config.dictConfig'
MANAGERS	[]（空列表）
MEDIA_ROOT	''（空字符串）
MEDIA_URL	''（空字符串）
MIDDLEWARE_CLASSES	('django.middleware.common.CommonMiddleware', 'django.middleware.csrf.CsrfViewMiddleware')
MIGRATION_MODULES	{}（空字典）
MONTH_DAY_FORMAT	'F j'
NUMBER_GROUPING	0
PREPEND_WWW	False
ROOT_URLCONF	未定义
SECRET_KEY	''（空字符串）
SECURE_BROWSER_XSS_FILTER	False
SECURE_CONTENT_TYPE_NOSNIFF	False
SECURE_HSTS_INCLUDE_SUBDOMAINS	False
SECURE_HSTS_SECONDS	0
SECURE_PROXY_SSL_HEADER	None
SECURE_REDIRECT_EXEMPT	[]（空列表）
SECURE_SSL_HOST	None

（续）

设置	默认值
SECURE_SSL_REDIRECT	False
SERIALIZATION_MODULES	未定义
SERVER_EMAIL	'root@localhost'
SHORT_DATE_FORMAT	m/d/Y（如 12/31/2003）
SHORT_DATETIME_FORMAT	m/d/Y P（如 12/31/2003 4 p.m.）
SIGNING_BACKEND	'django.core.signing.TimestampSigner'
SILENCED_SYSTEM_CHECKS	[]（空列表）
TEMPLATES	[]（空列表）
TEMPLATE_DEBUG	False
TEST_RUNNER	'django.test.runner.DiscoverRunner'
TEST_NON_SERIALIZED_APPS	[]（空列表）
THOUSAND_SEPARATOR	,（逗号）
TIME_FORMAT	'P'（如 4 p.m.）
TIME_INPUT_FORMATS	('%H:%M:%S', '%H:%M:%S.%f', '%H:%M',)
TIME_ZONE	'America/Chicago'
USE_ETAGS	False
USE_I18N	True
USE_L10N	False
USE_THOUSAND_SEPARATOR	False
USE_TZ	False
USE_X_FORWARDED_HOST	False
WSGI_APPLICATION	None
YEAR_MONTH_FORMAT	'F Y'
X_FRAME_OPTIONS	'SAMEORIGIN'

D.8.2 身份验证设置

表 D-2：Django 身份验证设置

设置	默认值
AUTHENTICATION_BACKENDS	'django.contrib.auth.backends.ModelBackend'
AUTH_USER_MODEL	'auth.User'
LOGIN_REDIRECT_URL	'/accounts/profile/'
LOGIN_URL	'/accounts/login/'
LOGOUT_URL	'/accounts/logout/'
PASSWORD_RESET_TIMEOUT_DAYS	3
PASSWORD_HASHERS	('django.contrib.auth.hashers.PBKDF2PasswordHasher', 'django.contrib.auth.hashers.PBKDF2SHA1PasswordHasher', 'django.contrib.auth.hashers.BCryptPasswordHasher', 'django.contrib.auth.hashers.SHA1PasswordHasher', 'django.contrib.auth.hashers.MD5PasswordHasher', 'django.contrib.auth.hashers.UnsaltedMD5PasswordHasher', 'django.contrib.auth.hashers.CryptPasswordHasher')

D.8.3 消息设置

表 D-3：Django 消息设置

设置	默认值
MESSAGE_LEVEL	messages.INFO
MESSAGE_STORAGE	'django.contrib.messages.storage.fallback.FallbackStorage'
MESSAGE_TAGS	{messages.DEBUG: 'debug', messages.INFO: 'info', messages.SUCCESS: 'success', messages.WARNING: 'warning', messages.ERROR: 'error'}

D.8.4 会话设置

表 D-4：Django 会话设置

设置	默认值
SESSION_CACHE_ALIAS	default
SESSION_COOKIE_AGE	1209600（2 周对应的秒数）
SESSION_COOKIE_DOMAIN	None
SESSION_COOKIE_HTTPONLY	True
SESSION_COOKIE_NAME	'sessionid'

（续）

设置	默认值
SESSION_COOKIE_PATH	'/'
SESSION_COOKIE_SECURE	False
SESSION_ENGINE	'django.contrib.sessions.backends.db'
SESSION_EXPIRE_AT_BROWSER_CLOSE	False
SESSION_FILE_PATH	None
SESSION_SAVE_EVERY_REQUEST	False
SESSION_SERIALIZER	'django.contrib.sessions.serializers.JSONSerializer'

D.8.5 Django 网站设置

表 D-5：Django 网站设置

设置	默认值
SITE_ID	未定义

D.8.6 静态文件设置

表 D-6：Django 静态文件设置

设置	默认值
STATIC_ROOT	None
STATIC_URL	None
STATICFILES_DIRS	[]（空列表）
STATICFILES_STORAGE	'django.contrib.staticfiles.storage.StaticFilesStorage'
STATICFILES_FINDERS	("django.contrib.staticfiles.finders.FileSystemFinder", "django.contrib.staticfiles.finders.AppDirectoriesFinder")

附录 E 内置模板标签和过滤器

第 3 章介绍了一些最有用的内置模板标签和过滤器。然而，Django 自带的内置标签和过滤器还有很多。这篇附录概述 Django 中可用的全部模板标签和过滤器。详细说明和使用举例参见 Django 文档。

E.1 内置标签

E.1.1 autoescape

控制当前的自动转义行为。这个标签的参数为 on 或 off，指明在块中自动转义是否生效。块以 endautoescape 标签结尾。

启用自动转义时，包含 HTML 的变量先转义再输出（在此之前先应用过滤器）。这与手动在各个变量上应用 escape 过滤器效果一样。

唯一的例外是被生成变量的代码或者通过 safe 或 escape 过滤器标记为无需转义的安全内容。

用法举例：

```
{% autoescape on %}
    {{ body }}
{% endautoescape %}
```

E.1.2 block

定义可由子模板覆盖的块。详情参见 3.11 节。

E.1.3 comment

忽略 {% comment %} 和 {% endcomment %} 之间的全部内容。起始标签可以添加可选的注解，例如说明为何要注释掉代码。

comment 标签不能嵌套。

E.1.4 csrf_token

这个标签提供 CSRF 防护。关于跨站请求伪造（Cross Site Request Forgery）的详细信息参见第 3 章和第 19 章。

E.1.5 cycle

每执行这个标签一次输出参数中的一个值。第一次执行输出第一个参数，第二次执行输出第二个参数，以此类推。所有参数都输出一遍之后，回到第一个参数，再依次输出。这个标签在循环中特别有用：

```
{% for o in some_list %}
```

```
        <tr class="{% cycle 'row1' 'row2' %}">
            ...
        </tr>
    {% endfor %}
```

第一次迭代生成的 HTML 使用 row1 类，第二次迭代使用 row2 类，第三次迭代使用 row1 类，以此类推。参数的值也可以通过变量指定。假如有两个模板变量 rowvalue1 和 rowvalue2，可以像这样在二者的值之间切换：

```
    {% for o in some_list %}
        <tr class="{% cycle rowvalue1 rowvalue2 %}">
            ...
        </tr>
    {% endfor %}
```

此外，也可以混用变量和字符串：

```
    {% for o in some_list %}
        <tr class="{% cycle 'row1' rowvalue2 'row3' %}">
            ...
        </tr>
    {% endfor %}
```

一个 cycle 标签中可以使用任意多个值（以空格分开）。放在单引号（'）或双引号（"）中的值视作字符串字面量，没有引号的视为模板变量。

E.1.6 debug

输出大量调试信息，包括当前上下文和导入的模块。

E.1.7 extends

表明所在模板扩展某个父模板。这个标签有两种用法：

1. {% extends "base.html" %}（有引号），以字面值 "base.html" 为父模板的名称。
2. {% extends variable %} 以 variable 的值为父模板的名称。如果 variable 的求值结果是一个字符串，Django 以那个字符串为父模板的名称；如果求值结果是一个 Template 对象，Django 使用那个对象为父模板。

E.1.8 filter

使用一个或多个过滤器过滤内容块。Django 中可用的过滤器参见 E.2 节。

E.1.9 firstof

输出参数中第一个不为 False 的变量值。如果传入的变量都为 False，不输出任何内容。用法示例：

```
    {% firstof var1 var2 var3 %}
```

等效于：

```
    {% if var1 %}
```

```
    {{ var1 }}
{% elif var2 %}
    {{ var2 }}
{% elif var3 %}
    {{ var3 }}
{% endif %}
```

E.1.10 for

迭代数组中的各个元素，并把当前迭代的元素赋值给一个上下文变量。例如，下述代码列出 athlete_list 中的各位运动员：

```
<ul>
{% for athlete in athlete_list %}
    <li>{{ athlete.name }}</li>
{% endfor %}
</ul>
```

可以使用 {% for obj in list reversed %} 反向迭代列表。如果需要迭代列表构成的列表，可以把各个子列表拆包出来，赋予单独的变量。访问字典中的元素时也可以这么做。假如上下文中有个名为 data 的字典，下述代码显示它的键和值：

```
{% for key, value in data.items %}
    {{ key }}: {{ value }}
{% endfor %}
```

E.1.11 for … empty

for 标签有个可选的 {% empty %} 子句，在指定的数组为空或无法找到时显示一些文本。

```
<ul>
{% for athlete in athlete_list %}
    <li>{{ athlete.name }}</li>
{% empty %}
    <li>Sorry, no athletes in this list.</li>
{% endfor %}
</ul>
```

E.1.12 if

{% if %} 计算变量的值，如果为真（即存在、不为空、不是假值），输出块中的内容：

```
{% if athlete_list %}
    Number of athletes: {{ athlete_list|length }}
{% elif athlete_in_locker_room_list %}
    Athletes should be out of the locker room soon!
{% else %}
    No athletes.
{% endif %}
```

在上述代码中，如果 athlete_list 不为空，{{ athlete_list|length }} 将显示运动员的数量。可以看出，if 标签可以有一个或多个 {% elif %} 子句，以及一个 {% else %} 子句，在前述条件判断都失败时显示。这些子

句都是可选的。

布尔运算符

if 标签可以使用 and、or 或 not 测试多个变量或取反指定的变量：

```
{% if athlete_list and coach_list %}
    Both athletes and coaches are available.
{% endif %}

{% if not athlete_list %}
    There are no athletes.
{% endif %}

{% if athlete_list or coach_list %}
    There are some athletes or some coaches.
{% endif %}
```

同一个标签中可以同时使用 and 和 or，前者的优先级较高。例如：

```
{% if athlete_list and coach_list or cheerleader_list %}
```

是这样解释的：

```
if (athlete_list and coach_list) or cheerleader_list
```

但是，在 if 标签中使用括号是无效的句法。如果想指明优先级，应该使用嵌套的 if 标签。

if 标签还可以使用 ==、!=、<、>、<=、>= 和 in 等运算符，各自的作用参见表 E-1。

表 E-1：模板标签中的布尔运算符

运算符	示例
==	{% if somevar == x %} …
!=	{% if somevar != x %} …
<	{% if somevar < 100 %} …
>	{% if somevar > 10 %} …
<=	{% if somevar <= 100 %} …
>=	{% if somevar >= 10 %} …
in	{% if bc in abcdef %}

复杂的表达式

上述运算符可以结合起来构成复杂的表达式。此时，要知道表达式是如何计算的，即优先级规则。这些运算符的优先级从低到高如下所示：

- or
- and

- `not`
- `in`
- `==`，`!=`，`<`，`>`，`<=`，`>=`

这与 Python 中的优先级完全一样。

过滤器

在 `if` 表达式中还可以使用过滤器。例如：

```
{% if messages|length >= 100 %}
You have lots of messages today!
{% endif %}
```

E.1.13 ifchanged

检查前一次迭代之后值是否变了。`{% ifchanged %}` 块标签在循环中使用。有两种用法：

1. 检查渲染得到的内容，与之前的状态比较，只在有变化时显示内容。
2. 检查指定的一个或多个变量的值是否有变化。

E.1.14 ifequal

两个参数相等时输出块中的内容。例如：

```
{% ifequal user.pk comment.user_id %}
    ...
{% endifequal %}
```

可用 `if` 标签和 `==` 运算符代替 `ifequal` 标签。

E.1.15 ifnotequal

与 `ifequal` 类似，不过测试的是两个参数是否不等。可用 `if` 和 `!=` 运算符代替。

E.1.16 include

加载一个模板，在当前上下文中渲染。这是在模板中引入其他模板的方式。模板的名称可以是一个变量：

```
{% include template_name %}
```

或者硬编码的字符串（有引号）：

```
{% include "foo/bar.html" %}
```

E.1.17 load

加载自定义的模板标签。例如，下述模板代码加载 `somelibrary` 和 `package` 包中 `otherlibrary` 注册的全部标签和过滤器：

```
{% load somelibrary package.otherlibrary %}
```

此外，还可以使用 from 参数有选择地从库中加载具体的过滤器和标签。

下述示例从 somelibrary 中加载名为 foo 和 bar 的标签或过滤器：

```
{% load foo bar from somelibrary %}
```

详情参见 8.7 节。

E.1.18 lorem

显示随机的占位拉丁文本。可用于为模板提供示例数据。用法：

```
{% lorem [count] [method] [random] %}
```

{% lorem %} 标签可接受零个、一个、两个或三个参数。这些参数分别是：

1. count：一个数字（或变量），指定生成的段落或单词数量（默认为 1）。
2. method：为表示单词的 w、表示 HTML 段落的 p 或表示纯文本段落的 b（默认为 b）。
3. random：生成文本时随机输出（指定时），不使用常见的段落（Lorem ipsum dolor sit amet...）。

例如，{% lorem 2 w random %} 随机输出两个拉丁单词。

E.1.19 now

以参数指定的格式显示当前日期和（或）时间。参数中可用的格式说明符参见 date 过滤器。例如：

```
It is {% now "jS F Y H:i" %}
```

传入的格式也可以是某个预定义的值，如 DATE_FORMAT、DATETIME_FORMAT、SHORT_DATE_FORMAT 或 SHORT_DATETIME_FORMAT。预定义格式的输出内容根据当前本地化设置和是否启用本地化格式而有所不同。例如：

```
It is {% now "SHORT_DATETIME_FORMAT" %}
```

E.1.20 regroup

按共有属性重新分组一组相似的对象。

{% regroup %} 生成分组对象构成的列表。每个分组对象有两个属性：

- grouper：分组的依据（如字符串 "India" 或 "Japan"）
- list：分组中的元素列表（如国家为 "India" 的城市列表）

注意，{% regroup %} 不排序输入！

任何有效的模板查找对象都是有效的分组属性，包括方法、属性、字典的键和列表的元素。

E.1.21 spaceless

删除 HTML 标签之间的空白，包括制表符和换行符。用法示例：

```
{% spaceless %}
    <p>
```

```
        <a href="foo/">Foo</a>
    </p>
{% endspaceless %}
```

这个示例返回的 HTML 如下：

```
<p><a href="foo/">Foo</a></p>
```

E.1.22 templatetag

输出构成模板标签句法的字符。模板系统自身无法转义，因此若想显示构成模板标签的字符，必须使用 {% templatetag %} 标签。参数指明输出模板标签的哪一部分：

- openblock 输出 {%
- closeblock 输出 %}
- openvariable 输出 {{
- closevariable 输出 }}
- openbrace 输出 {
- closebrace 输出 }
- opencomment 输出 {#
- closecomment 输出 #}

用法示例：

```
{% templatetag openblock %} url 'entry_list' {% templatetag closeblock %}
```

E.1.23 url

返回指定视图函数和可选参数对应的绝对路径（不带域名的 URL）。路径中的特殊字符会使用 iri_to_uri() 编码。使用这种方式输出链接符合 DRY 原则，因为无需在模板中硬编码 URL。

```
{% url 'some-url-name' v1 v2 %}
```

第一个参数是视图函数的路径，格式为 package.package.module.function，可以是放在引号里的字面量，也可以是上下文变量。余下的参数是可选的，以空格分开，用于指定 URL 中的参数。

E.1.24 verbatim

不让模板引擎渲染块中的内容。JavaScript 模板层经常使用这个标签防止与 Django 的句法冲突。

E.1.25 widthratio

插入条形图等图像时用于计算上至某个值的长宽比，然后把这一比例规整。例如：

```
<img src="bar.png" alt="Bar" height="10" width="{% widthratio this_value max_value max_width
%}" />
```

E.1.26 with

把复杂的变量缓存到一个简单的名称下。多次访问耗资源的（如访问数据库）方法时可以这么做。例如：

```
{% with total=business.employees.count %}
    {{ total }} employee{{ total|pluralize }}
{% endwith %}
```

E.2 内置过滤器

E.2.1 add

把参数加到值上。例如：

```
{{ value|add:"2" }}
```

如果 `value` 为 `4`，输出 `6`。

E.2.2 addslashes

在引号前加上斜线。可用于转义 CSV 中的字符串。例如：

```
{{ value|addslashes }}
```

如果 `value` 为 `"I'm using Django"`，输出 `"I\'m using Django"`。

E.2.3 capfirst

把值的首字母变成大写。如果第一个字符不是字母，这个过滤器没有效果。

E.2.4 center

在指定宽度中居中显示值。例如：

```
"{{ value|center:"14" }}"
```

如果 `value` 为 `"Django"`，输出 `"␣␣␣␣Django␣␣␣␣"`（␣ 表示一个空格）。

E.2.5 cut

从字符串中删除参数指定的值。

E.2.6 date

使用指定的格式格式化日期。格式与 PHP 的 `date()` 函数类似，不过有些区别。

> **提示**
>
> 这些格式字符只能在 Django 的模板中使用，设计时考虑到便于设计师迁移，因此与 PHP 采用的格式兼容。完整的格式字符串参见 Django 文档。

例如：

```
{{ value|date:"D d M Y" }}
```

如果 value 是一个 datetime 对象（例如 datetime.datetime.now() 的结果），输出的字符串类似于 "Fri 01 Jul 2016"。也可以传入预定义的格式 DATE_FORMAT、DATETIME_FORMAT、SHORT_DATE_FORMAT 或 SHORT_DATETIME_FORMAT，以及使用日期格式说明符自定义的格式。

E.2.7 default

如果 value 是假值，使用指定的默认值；否则，使用 value 的值。例如：

```
{{ value|default:"nothing" }}
```

E.2.8 default_if_none

当且仅当 value 为 None 时使用指定的默认值，否则使用 value 的值。

E.2.9 dictsort

按参数指定的键排序字典构成的列表，返回排序后的列表。例如：

```
{{ value|dictsort:"name" }}
```

E.2.10 dictsortreversed

按参数指定的键排序字典构成的列表，返回反向排序后的列表。

E.2.11 divisibleby

如果值能被参数整除，返回 True。例如：

```
{{ value|divisibleby:"3" }}
```

如果 value 为 21，返回 True。

E.2.12 escape

转义字符串中的 HTML。具体而言，是做下述替换：

- < 转换成 <
- > 转换成 >
- '（单引号）转换成 '
- "（双引号）转换成 "
- & 转换成 &

输出字符串时才做转义，因此在一串过滤器中 escape 放在什么位置没关系，它始终作为最后一个过滤器应用。

E.2.13 escapejs

转义 JavaScript 字符串中的字符。这个过滤器得到的结果不能在 HTML 中安全使用，但是使用模板生成 JavaScript/JSON 时能防止句法出错。

E.2.14 filesizeformat

把值格式化为人类可读的文件大小（如 '13 KB'、'4.1 MB'、'102 bytes'，等等）。例如：

```
{{ value|filesizeformat }}
```

如果 value 为 123456789，输出 117.7 MB。

E.2.15 first

返回列表中的第一个元素。

E.2.16 floatformat

不指定参数时，近似浮点数，只保留一位小数（前提是有小数）。如果指定了整数参数，floatformat 近似时保留相应位数的小数。

如果 value 为 34.23234，{{ value|floatformat:3 }} 输出 34.232。

E.2.17 get_digit

从一个数字中取回指定的那一位，1 表示最低位。

E.2.18 iriencode

把 IRI（Internationalized Resource Identifier，国际化资源标识符）转换成适合在 URL 中使用的字符串。

E.2.19 join

使用字符串连接列表，类似于 Python 的 str.join(list)。

E.2.20 last

返回列表中的最后一个元素。

E.2.21 length

返回值的长度。字符串和列表都适用。

E.2.22 length_is

如果值的长度与参数相等，返回 True，否则返回 False。例如：

```
{{ value|length_is:"4" }}
```

E.2.23 linebreaks

把纯文本中的换行替换成适当的 HTML 标签。一个换行符替换成一个 HTML `<br />` 标签，换行之后还有新行则替换成段落结束标签（`</p>`）。

E.2.24 linebreaksbr

把纯文本中的所有换行都替换成 HTML `<br />` 标签。

E.2.25 linenumbers

显示带有行号的文本。

E.2.26 ljust

在指定宽度中左对齐值。例如：

```
{{ value|ljust:"10" }}
```

如果 value 为 "Django"，输出 "Django␣␣␣␣"（␣ 表示一个空格）。

E.2.27 lower

把字符串转换成全小写形式。

E.2.28 make_list

把值转换成列表。值为字符串时，返回各字母构成的列表。值为整数时，先把参数转换成 Unicode 字符串，然后再创建列表。

E.2.29 phone2numeric

把电话号码（可能包含字母）转换成等效的数值。输入无需是有效的电话号码，任何字符串都能转换。例如：

```
{{ value|phone2numeric }}
```

如果 value 为 800-COLLECT，输出 800-2655328。

E.2.30 pluralize

值不为 1 时返回复数后缀。默认后缀为 "s"。

单词复数变形较为复杂时，可以指定单数和复数后缀，以逗号分开。例如：

```
You have {{ num_cherries }} cherr{{ num_cherries|pluralize:"y,ies" }}.
```

E.2.31 pprint

对 pprint.pprint() 的包装，用于调试。

E.2.32 random

返回指定列表中的一个随机元素。

E.2.33 rjust

在指定宽度中右对齐值。例如：

```
{{ value|rjust:"10" }}
```

如果 value 为 "Django"，输出 "␣␣␣␣Django"（␣ 表示一个空格）。

E.2.34 safe

标记字符串在输出之前无需进一步转义 HTML。关闭自动转义时这个过滤器没有效果。

E.2.35 safeseq

把 safe 过滤器应用到序列中的各个元素上。可以与其他作用于序列的过滤器合用（如 join）。例如：

```
{{ some_list|safeseq|join:", " }}
```

此时不能直接使用 safe 过滤器，因为它先把变量转换成字符串，而不是处理序列中的单个元素。

E.2.36 slice

返回列表的切片。句法与 Python 的列表切片一样。

E.2.37 slugify

转换成 ASCII。把空格转换成连字符。把字母、数字、下划线和连字符之外的字符删除。转换成小写。去掉首尾的空白。

E.2.38 stringformat

根据参数指定的格式说明符格式化变量。说明符使用 Python 字符串格式化句法，但是不用前导 %。

E.2.39 striptags

尽全力去除 (X)HTML 标签。例如：

```
{{ value|striptags }}
```

E.2.40 time

使用指定格式格式化时间。与 date 过滤器一样，可以是预定义的 TIME_FORMAT，也可以是自定义的格式。

E.2.41 timesince

把日期格式化为距某一刻的时间（如"4 days, 6 hours"）。有个可选的参数，指定比较的日期基点（如不指

定，与 now 比较）。

E.2.42 timeuntil

距指定日期或日期时间的跨度。

E.2.43 title

把字符串转换成标题格式，即单词首字母大写，其余字母小写。

E.2.44 truncatechars

字符串的字符串超过指定长度时截断。截断后的字符串以可翻译的省略号（…）结尾。例如：

```
{{ value|truncatechars:9 }}
```

E.2.45 truncatechars_html

与 truncatechars 类似，不过知道如何截断 HTML 标签。

E.2.46 truncatewords

在指定的单词个数后截断字符串。

E.2.47 truncatewords_html

与 truncatewords 类似，不过知道如何截断 HTML 标签。

E.2.48 unordered_list

递归遍历嵌套的列表，返回一个 HTML 无序列表（不含起始和结束标签）。

E.2.49 upper

把字符串转换成全大写字母形式。

E.2.50 urlencode

转义，以便在 URL 中使用。

E.2.51 urlize

把文本中的 URL 和电子邮件地址转换成可点击的链接。支持以 http://、https:// 或 www. 开头的 URL。

E.2.52 urlizetrunc

与 urlize 一样，把 URL 和电子邮件地址转换成可点击的链接，但是在指定的字符长度处截断 URL。例如：

```
{{ value|urlizetrunc:15 }}
```

如果 value 为 "Check out www.djangoproject.com"，输出 "Check out www.djangopr…"。与 urlize 一样，这个过滤器只应该应用于纯文本。

E.2.53 wordcount

返回字数。

E.2.54 wordwrap

在指定的长度处换行。

E.2.55 yesno

把 True、False 和 None（可选）映射到字符串"yes"、"no"、"maybe"上，或者是通过参数传入的列表上（以逗号分隔），根据真假值情况，返回相应的字符串。例如：

```
{{ value|yesno:"yeah,no,maybe" }}
```

E.3 国际化标签和过滤器

Django 提供了能在模板中控制国际化行为的标签和过滤器。使用这些标签和过滤器可以细致控制翻译、格式化和时区转换。

E.3.1 i18n

这个库用于指定模板中可翻译的文本。若想使用这个库，把 USE_I18N 设为 True，然后使用 {% load i18n %} 加载。

E.3.2 l10n

这个库用于本地化模板中的文本。为此，只需使用 {% load l10n %} 加载这个库，不过通常都会把 USE_L10N 设为 True，因此本地化默认已启用。

E.3.3 tz

这个库的作用是在模板中控制时区转换。与 l10n 类似，只需使用 {% load tz %} 加载这个库，不过通常都会把 USE_TZ 设为 True，因此默认就会转换成当地时间。参见文档。

E.4 其他标签和过滤器库

E.4.1 static

为了链接保存在 STATIC_ROOT 中的静态文件，Django 提供了 static 模板标签。不管用不用 RequestContext，都可以使用这个标签。

```
{% load static %}
```

```
<img src="{% static "images/hi.jpg" %}" alt="Hi!" />
```

此外，还可以使用上下文变量。假如向模板传入了 user_stylesheet 变量：

```
{% load static %}
<link rel="stylesheet" href="{% static user_stylesheet %}" type="text/css" media="screen" />
```

如果只想获取静态文件的 URL，而不显示，可以使用稍微不同的调用方式：

```
{% load static %}
{% static "images/hi.jpg" as myphoto %}
<img src="{{ myphoto }}"></img>
```

staticfiles 应用也提供了 static 标签，它使用 staticfiles 的 STATICFILES_STORAGE 设置构建指定路径的 URL（而不是仅仅把 STATIC_URL 设置和指定路径传给 urllib.parse.urljoin()）。如果使用云服务伺服静态文件，要使用这个标签。例如：

```
{% load static from staticfiles %}
<img src="{% static "images/hi.jpg" %}" alt="Hi!" />
```

E.4.2 get_static_prefix

多数情况下，应该使用 Django 自带的 static 标签；如果想更好地控制在何处以及如何插入 STATIC_URL，可以使用 get_static_prefix 标签。

```
{% load static %}
<img src="{% get_static_prefix %}images/hi.jpg" alt="Hi!" />
```

如果需要多次使用这个值，可以像下面这样编写代码，避免额外消耗：

```
{% load static %}
{% get_static_prefix as STATIC_PREFIX %}

<img src="{{ STATIC_PREFIX }}images/hi.jpg" alt="Hi!" />
<img src="{{ STATIC_PREFIX }}images/hi2.jpg" alt="Hello!" />
```

E.4.3 get_media_prefix

与 get_static_prefix 类似，get_media_prefix 获取媒体文件前缀 MEDIA_URL。例如：

```
<script type="text/javascript" charset="utf-8">
var media_path = '{% get_media_prefix %}';
</script>
```

此外，Django 还提供了其他模板标签库，若想使用，要在 INSTALLED_APPS 设置中列出，然后在模板中使用 {% load %} 标签加载。

附录 F 请求和响应对象

Django 使用请求和响应对象在系统中传递状态。

请求一个页面时，Django 创建一个 HttpRequest 对象，附带请求的元数据。然后 Django 加载适当的视图，把 HttpRequest 对象作为第一个参数传给视图函数。视图则返回一个 HttpResponse 对象。

本篇附录说明 HttpRequest 和 HttpResponse 对象的 API。二者在 django.http 模块中定义。

F.1 HttpRequest 对象

F.1.1 属性

如未说明，应该把属性视为只读的。session 是个例外。

HttpRequest.scheme

一个字符串，表示请求模式（通常是 http 或 https）。

HttpRequest.body

HTTP 请求的原始主体，类型为字节字符串。原始主体可用于处理 HTML 之外的格式，如图像、XML 载荷等。常规的表单数据使用 HttpRequest.POST 处理。

还可以使用类似文件的接口读取 HttpRequest。参见 HttpRequest.read()。

HttpRequest.path

一个字符串，表示所请求页面的完整路径（不含域名）。

例如：/music/bands/the_beatles/。

HttpRequest.path_info

在某些 Web 服务器配置下，URL 中主机名后面的部分分成脚本前缀和路径信息。不管使用哪个 Web 服务器，path_info 属性的值始终是路径信息部分。在代码中用 path_info 代替 path 便于在测试和线上服务器之间切换。

例如，如果把应用程序的 WSGIScriptAlias 设为 /minfo，那么 path 的是可能是 /minfo/music/bands/the_beatles/，而 path_info 的值则是 /music/bands/the_beatles/。

HttpRequest.method

一个字符串，表示发起请求所用的 HTTP 方法。值始终为大写形式。例如：

```
if request.method == 'GET':
    do_something()
```

```python
    elif request.method == 'POST':
        do_something_else()
```

HttpRequest.encoding

一个字符串，表示用于解码表单数据的编码（或者为 None，表示使用 DEFAULT_CHARSET 设置）。访问表单数据时可以重设这个属性的值。这样，后续对属性的访问（例如读取 GET 或 POST 数据）都使用新设的 encoding 值。如果知道表单数据的编码与 DEFAULT_CHARSET 不同，可以这么做。

HttpRequest.GET

一个类似字典的对象，包含所有 HTTP GET 参数。参见 F.2 节。

HttpRequest.POST

一个类似字典的对象，包含所有 HTTP POST 参数（前提是请求中有表单数据）。参见 F.2 节。

如果想访问请求中的原始数据或非表单数据，通过 HttpRequest.body 属性访问。

POST 请求可以发送空的 POST 字典。例如，通过 HTTP POST 方法请求表单，但是不包含任何表单数据。因此不能使用 if request.POST 检测是不是 POST 方法；正确的做法是使用 if request.method == 'POST'（参见前文）。

注意，POST 字典中没有文件上传信息。参见 FILES。

HttpRequest.COOKIES

一个标准的 Python 字典，包含所有 cookie。键和值都是字符串。

HttpRequest.FILES

一个类似字典的对象，包含所有的上传文件。FILES 中的各个键是 <input type="file" name="" /> 标签 name 属性的值；值则是 UploadedFile 对象。

注意，只有请求方法为 POST，而且 <form> 标签设定了 enctype="multipart/form-data" 属性，FILES 中才有数据。否则，FILES 是个空的类似字典的对象。

HttpRequest.META

一个标准的 Python 字典，包含所有可用的 HTTP 首部。具体有哪些首部，取决于客户端和服务器，下面举几个例子：

- CONTENT_LENGTH：请求主体（字符串形式）的长度。
- CONTENT_TYPE：请求主体的 MIME 类型。
- HTTP_ACCEPT_ENCODING：可接受的响应编码。
- HTTP_ACCEPT_LANGUAGE：可接受的响应语言。
- HTTP_HOST：客户端发送的 HTTP Host 首部。
- HTTP_REFERER：前一个页面（如果有）。
- HTTP_USER_AGENT：客户端的用户代理字符串。
- QUERY_STRING：查询字符串，作为一个整体（未经解析）。

- REMOTE_ADDR：客户端的 IP 地址。
- REMOTE_HOST：客户端的主机名。
- REMOTE_USER：通过 Web 服务器验证身份的用户（如果有）。
- REQUEST_METHOD：一个字符串，如 "GET" 或 "POST"。
- SERVER_NAME：服务器的主机名。
- SERVER_PORT：服务器的端口（字符串）。

除了 CONTENT_LENGTH 和 CONTENT_TYPE 之外，请求的 HTTP 首部转换成 META 键的方式是，把所有字母变成大写，连字符替换成下划线，然后加上 HTTP_ 前缀。因此，X-Bender 首部在 META 属性中的键是 HTTP_X_BENDER。

HttpRequest.user

一个 AUTH_USER_MODEL 类型的对象，表示当前登录的用户。如果未登录，user 是 django.contrib.auth.models.AnonymousUser 的实例。可以使用 is_authenticated() 区分二者，例如：

```
if request.user.is_authenticated():
    # 针对已登录用户
else:
    # 针对匿名用户
```

只有激活了 AuthenticationMiddleware 才有 user 属性。

HttpRequest.session

一个类似字典的对象，可读可写，表示当前会话。只有启用了会话功能才有这个属性。

HttpRequest.urlconf

不是 Django 自己定义的，但是如果其他代码（如自定义的中间件类）设定了就可读取。有这个属性时，用作当前请求的根 URL 配置，把 ROOT_URLCONF 设置覆盖掉。

HttpRequest.resolver_match

一个 ResolverMatch 实例，表示解析出来的 URL。只有解析了 URL 才会设定这个属性，因此只在视图中可用，而不能在中间件方法中使用，因为中间件方法在解析 URL 之前执行（例如 process_request，此时可以使用 process_view 代替）。

F.1.2 方法

HttpRequest.get_host()

返回通过 HTTP_X_FORWARDED_HOST（如果启用了 USE_X_FORWARDED_HOST）和 HTTP_HOST 首部中的信息（以此顺序）确定的源主机。如果据此得不到值，这个方法按照 PEP 3333 所述的方式，把 SERVER_NAME 和 SERVER_PORT 连接在一起。

例如：127.0.0.1:8000。

注意

如果主机在多个代理后面，get_host() 方法无法返回值。一种解决方法是使用中间件重写代理首部，如下述示例所示：

```
class MultipleProxyMiddleware(object):
    FORWARDED_FOR_FIELDS = [
        'HTTP_X_FORWARDED_FOR',
        'HTTP_X_FORWARDED_HOST',
        'HTTP_X_FORWARDED_SERVER',
    ]

    def process_request(self, request):
        """
        Rewrites the proxy headers so that only the most
        recent proxy is used.
        """
        for field in self.FORWARDED_FOR_FIELDS:
            if field in request.META:
                if ',' in request.META[field]:
                    parts = request.META[field].split(',')
                    request.META[field] = parts[-1].strip()
```

这个中间件应该放在使用 get_host() 所得值的其他中间件前面，例如 CommonMiddleware 或 CsrfViewMiddleware。

HttpRequest.get_full_path()

返回 path，以及后面的查询字符串（如果有）。

例如：/music/bands/the_beatles/?print=true。

HttpRequest.build_absolute_uri(location)

返回 location 对应的绝对 URI。如果未指定 location，使用 request.get_full_path()。

如果 location 已经是绝对 URI，不再修改；否则使用请求中的服务器参数构建绝对 URI。

例如：http://example.com/music/bands/the_beatles/?print=true。

HttpRequest.get_signed_cookie()

返回指定签名 cookie 的值；如果签名失效，抛出 django.core.signing.BadSignature 异常。如果提供了 default 参数，不再抛出异常，而是返回指定的默认值。

可选的 salt 参数用于提供额外的防护措施，以免暴力攻击密钥。如果提供 max_age 参数，使用它检查 cookie 值的签名时间戳，确保 cookie 没有超过 max_age 指定的秒数。

例如：

```
>>> request.get_signed_cookie('name')
'Tony'
>>> request.get_signed_cookie('name', salt='name-salt')
'Tony' # 假设 cookie 是使用这个盐值设定的
>>> request.get_signed_cookie('non-existing-cookie')
...
```

```
KeyError: 'non-existing-cookie'
>>> request.get_signed_cookie('non-existing-cookie', False)
False
>>> request.get_signed_cookie('cookie-that-was-tampered-with')
...
BadSignature: ...
>>> request.get_signed_cookie('name', max_age=60)
...
SignatureExpired: Signature age 1677.3839159 > 60 seconds
>>> request.get_signed_cookie('name', False, max_age=60)
False
```

`HttpRequest.is_secure()`

为安全请求时，即通过 HTTPS 请求时，返回 `True`。

`HttpRequest.is_ajax()`

检查 `HTTP_X_REQUESTED_WITH` 首部中有没有字符串 `"XMLHttpRequest"`，判断请求是不是通过 `XMLHttpRequest` 发起的；如果是，返回 `True`。多数现代的 JavaScript 库都会发送这个首部。自己（在浏览器端）编写 `XMLHttpRequest` 调用时，若想让 `is_ajax()` 起作用，要手动设定这个首部。

如果响应根据请求是不是 Ajax 而有区别，而且使用了某种形式的缓存，如 Django 的缓存中间件，应该使用 `vary_on_headers('HTTP_X_REQUESTED_WITH')` 装饰视图，这样响应才能正确缓存。

`HttpRequest.read(size=None)`

`HttpRequest.readline()`

`HttpRequest.readlines()`

`HttpRequest.xreadlines()`

`HttpRequest.iter()`

这几个方法实现了类似文件的接口，用于从 `HttpRequest` 实例中读取数据。因此，它们能以流的形式处理入站请求。常见的使用场景是使用迭代的解析器处理大型 XML，这样无需在内存中构建整个 XML 树。

因为接口是标准的，所以可以直接把 `HttpRequest` 实例传给 XML 解析器，如 `ElementTree`：

```
import xml.etree.ElementTree as ET
for element in ET.iterparse(request):
    process(element)
```

F.2 QueryDict 对象

`HttpRequest` 对象的 `GET` 和 `POST` 属性是 `django.http.QueryDict` 的实例，这是一个自定义的类似字典的类，能处理一个键有多个值的情况。之所以要有这么一个类，是因为某些 HTML 表单元素，尤其是 `<select multiple>`，同一个键需要多个值。

在常规的请求-响应循环中，`request.POST` 和 `request.GET` 这两个 `QueryDict` 对象是不可变的。若想获得可变

的版本，使用 .copy()。

F.2.1 方法

QueryDict 实现了字典的所有标准方法，因为它是字典的子类。但是，下述方法的行为有所不同。

QueryDict.__init__()

根据 query_string 实例化 QueryDict 对象。

```
>>> QueryDict('a=1&a=2&c=3')
<QueryDict: {'a': ['1', '2'], 'c': ['3']}>
```

如果未传入 query_string，得到的 QueryDict 实例是空的（没有键也没有值）。

多数 QueryDict 对象，尤其是 request.POST 和 request.GET，是不可变的。如果自己实例化，可以把 mutable=True 传给 __init__()，创建一个可变的对象。

用于设定键和值的字符串会从 encoding 转换成 Unicode 编码。如果未设定编码，默认为 DEFAULT_CHARSET。

QueryDict.__getitem__(key)

返回指定键对应的值。如果有多个值，__getitem__() 返回最后一个值。如果键不存在，抛出 django.utils.datastructures.MultiValueDictKeyError。

QueryDict.__setitem__(key, value)

把指定键对应的值设为 [value]（一个 Python 列表，只有一个元素 value）。注意，这个方法与有副作用的其他字典函数一样，只能在可变的 QueryDict 对象（例如通过 copy() 创建的）上调用。

QueryDict.__contains__(key)

如果有指定的键，返回 True。有了这个方法，就可以这样判断：if "foo" in request.GET。

QueryDict.get(key, default)

与 __getitem__() 的逻辑一样，但是键不存在时，返回指定的默认值。

QueryDict.setdefault(key, default)

与标准的字典方法 setdefault() 类似，不过内部使用的是 __setitem__() 方法。

QueryDict.update(other_dict)

参数为 QueryDict 对象或标准的字典。与标准的字典方法 update() 类似，不过是追加到现有的元素中，而不是把元素替换掉。例如：

```
>>> q = QueryDict('a=1', mutable=True)
>>> q.update({'a': '2'})
>>> q.getlist('a')
['1', '2']
>>> q['a'] # 返回最后一个元素
['2']
```

`QueryDict.items()`

与标准的字典方法 `items()` 类似，不过与 `__getitem__()` 一样，值来自最后一个元素。例如：

```
>>> q = QueryDict('a=1&a=2&a=3')
>>> q.items()
[('a', '3')]
```

`QueryDict.iteritems()`

与标准的字典方法 `iteritems()` 类似，同样与 `QueryDict.__getitem__()` 一样，值来自最后一个元素。

`QueryDict.iterlists()`

类似于 `QueryDict.iteritems()`，不过包含字典中每个成员的所有值（构成一个列表）。

`QueryDict.values()`

与标准的字典方法 `values()` 类似，不过与 `QueryDict.__getitem__()` 一样，值来自最后一个元素。例如：

```
>>> q = QueryDict('a=1&a=2&a=3')
>>> q.values()
['3']
```

`QueryDict.itervalues()`

与 `QueryDict.values()` 类似，不过多了一个迭代器。

此外，`QueryDict` 还定义了下述方法。

`QueryDict.copy()`

使用 Python 标准库中的 `copy.deepcopy()` 创建 `QueryDict` 对象的副本。即便源对象是不可变的，得到的副本也是可变的。

`QueryDict.getlist(key, default)`

返回指定键对应的值（一个 Python 列表）。如果键不存在，而且没有提供默认值，返回一个空列表。除非默认值不是列表，否则始终返回列表。

`QueryDict.setlist(key, list)`

把指定的键对应的值设为指定的列表（不同于 `__setitem__()`）。

`QueryDict.appendlist(key, item)`

把元素追加到键对应的内部列表中。

`QueryDict.setlistdefault(key, default_list)`

类似于 `setdefault`，不过参数是一个列表，而不是单个值。

QueryDict.lists()

类似于 items()，不过包含每个字典成员的所有值（列表）。例如：

```
>>> q = QueryDict('a=1&a=2&a=3')
>>> q.lists()
[('a', ['1', '2', '3'])]
```

QueryDict.pop(key)

返回指定键对应的值（列表），并把它们从字典中删除。如果键不存在，抛出 KeyError。例如：

```
>>> q = QueryDict('a=1&a=2&a=3', mutable=True)
>>> q.pop('a')
['1', '2', '3']
```

QueryDict.popitem()

随意删除一个字典成员（因为字典是无序的），返回一个两元素元组，包含键和对应的值构成的列表。在空字典上调用，抛出 KeyError。例如：

```
>>> q = QueryDict('a=1&a=2&a=3', mutable=True)
>>> q.popitem()
('a', ['1', '2', '3'])
```

QueryDict.dict()

返回 QueryDict 对象的字典表示形式。QueryDict 对象中的每个 (key, list) 对在返回的字典中是 (key, item)，其中 item 是 list 中的最后一个元素（逻辑同 QueryDict.__getitem__()）。例如：

```
>>> q = QueryDict('a=1&a=3&a=5')
>>> q.dict()
{'a': '5'}
```

QueryDict.urlencode([safe])

返回查询字符串形式的字符串。例如：

```
>>> q = QueryDict('a=2&b=3&b=5')
>>> q.urlencode()
'a=2&b=3&b=5'
```

可以指定无需编码的字符。例如：

```
>>> q = QueryDict(mutable=True)
>>> q['next'] = '/a&b/'
>>> q.urlencode(safe='/')
'next=/a%26b/'
```

F.3 HttpResponse 对象

HttpRequest 对象由 Django 自动创建，而 HttpResponse 对象要由你自己创建。你编写的每个视图都要实例化、填充并返回一个 HttpResponse 对象。

HttpResponse 类在 django.http 模块中。

F.3.1 用法

传入字符串

常规的用法是把页面内容（字符串）传给 HttpResponse 构造方法：

```
>>> from django.http import HttpResponse
>>> response = HttpResponse("Here's the text of the Web page.")
>>> response = HttpResponse("Text only, please.", content_type="text/plain")
```

但是，如果想不断添加内容，可以把 response 视作类似文件的对象：

```
>>> response = HttpResponse()
>>> response.write("<p>Here's the text of the Web page.</p>")
>>> response.write("<p>Here's another paragraph.</p>")
```

传入迭代器

最后，还可以把一个迭代器传给 HttpResponse 构造方法。HttpResponse 将立即使用迭代器，以字符串形式存储内容，然后把迭代器丢掉。

如果想从迭代器中以流的形式把响应发给客户端，必须使用 StreamingHttpResponse 类。

设定首部字段

若想为响应设定首部字段，或者删除首部字段，把它视作一个字典：

```
>>> response = HttpResponse()
>>> response['Age'] = 120
>>> del response['Age']
```

注意，与字典不同，如果要删除的首部字段不存在，del 不抛出 KeyError。

设定 Cache-Control 和 Vary 首部字段推荐使用 django.utils.cache 中的 patch_cache_control() 和 patch_vary_headers() 方法，因为这两个字段可以有多个以逗号分开的值。这两个方法能确保其他值（例如中间件添加的）不被删除。

HTTP 首部字段不能换行。如果包含换行符（CR 或 LF），抛出 BadHeaderError。

让浏览器把响应视作文件附件

若想让浏览器把响应视作文件附件，提供 content_type 参数，并设定 Content-Disposition 首部。例如，下述代码返回一个 Microsoft Excel 电子表格：

```
>>> response = HttpResponse(my_data, content_type='application/vnd.ms-excel')
>>> response['Content-Disposition'] = 'attachment; filename="foo.xls"'
```

Content-Disposition 首部的值在 Django 中没什么特殊之处，但是容易忘记句法，因此我们举了个例子。

F.3.2 属性

`HttpResponse.content`

一个字节字符串，表示响应的内容；如果必要，编码为 Unicode 对象。

`HttpResponse.charset`

一个字符串，表示响应的编码字符集。如果实例化 `HttpResponse` 对象时没有指定，从 `content_type` 中提取；如果提取失败，使用 `DEFAULT_CHARSET` 设置。

`HttpResponse.status_code`

响应的 HTTP 状态码。

`HttpResponse.reason_phrase`

响应的状态描述短语。

`HttpResponse.streaming`

始终为 `False`。

这个属性存在的目的是让中间件把流式响应与常规响应区分开。

`HttpResponse.closed`

响应关闭后为 `True`。

F.3.3 方法

`HttpResponse.__init()__`

`HttpResponse.__init__(content='', content_type=None, status=200, reason=None, charset=None)`

使用指定的内容和内容类型实例化一个 `HttpResponse` 对象。`content` 为一个迭代器或一个字符串。为迭代器时，迭代器应该返回字符串，这些字符串连接在一起构成响应的内容。如果不是迭代器或字符串，访问时将被转换成字符串。后四个参数为：

- `content_type`：MIME 类型，可从字符集编码中推知，用于设定 HTTP `Content-Type` 首部。如未指定，由 `DEFAULT_CONTENT_TYPE` 和 `DEFAULT_CHARSET` 设置构成，默认为 `text/html; charset=utf-8`。
- `status`：响应的 HTTP 状态码。
- `reason`：响应的状态描述短语。如未提供，使用默认的短语。
- `charset`：编码响应所用的字符集。如未指定，从 `content_type` 中提取；如果提取失败，使用 `DEFAULT_CHARSET` 设置。

`HttpResponse.__setitem__(header, value)`

把指定的首部设为指定的值。`header` 和 `value` 都应该为字符串。

`HttpResponse.__delitem__(header)`

删除指定的首部。如果要删除的首部不存在，静默。不区分大小写。

`HttpResponse.__getitem__(header)`

返回指定首部的值。不区分大小写。

`HttpResponse.has_header(header)`

检查指定的首部（不区分大小写）是否存在，存在时返回 `True`，否则返回 `False`。

`HttpResponse.setdefault(header, value)`

设定一个首部，前提是那个首部尚未设定。

`HttpResponse.set_cookie()`

`HttpResponse.set_cookie(key, value='', max_age=None, expires=None, path='/', domain=None, secure=None, httponly=False)`

设定一个 cookie。参数与 Python 标准库中的 `Morsel` cookie 对象一样。

- `max_age` 是表示秒数的数字，或者为 `None`（默认值），表示 cookie 存在的时间与客户端浏览器会话一样长。如未指定 `expires`，据此计算。
- `expires` 是一个字符串，格式为 `"Wdy, DD-Mon-YY HH:MM:SS GMT"`，或者一个 UTC 时区的 `datetime.datetime` 对象。如果 `expires` 的值是 `datetime` 对象，将据此计算 `max_age`。
- `domain` 用于设定跨域 cookie。例如，设定 `domain=".lawrence.com"` 时，www.lawrence.com、blogs.lawrence.com 和 calendars.lawrence.com 等域名能访问 cookie。否则，只有设定 cookie 的域名才能读取 cookie。
- 如果不想让客户端 JavaScript 访问 cookie，设定 `httponly=True`。`HTTPOnly` 是 HTTP 响应首部 `Set-Cookie` 中的一个旗标，但不在 cookie 标准 RFC 2109 中，也不是所有浏览器都遵守。然而，如果浏览器遵守 `HTTPOnly` 规则，能降低客户端脚本访问受保护的 cookie 数据引发的风险。

`HttpResponse.set_signed_cookie()`

与 `set_cookie()` 类似，不过在设定之前会加密签名 cookie。与 `HttpRequest.get_signed_cookie()` 配对使用。可以使用可选的 `salt` 参数增加键的强度，但是要记得把 `salt` 值传给 `HttpRequest.get_signed_cookie()`。

`HttpResponse.delete_cookie()`

删除指定键对应的 cookie。如果要删除的键不存在，静默。

鉴于 cookie 的工作方式，`path` 和 `domain` 的值应该与传给 `set_cookie()` 的一样，否则无法删除 cookie。

`HttpResponse.write(content)`

`HttpResponse.flush()`

`HttpResponse.tell()`

这三个方法为 `HttpResponse` 对象实现类似文件的接口。作用与 Python 自带的相应方法一样。

`HttpResponse.getvalue()`

返回 `HttpResponse.content` 的值。这个方法把 `HttpResponse` 实例当做流式对象。

`HttpResponse.writable()`

始终为 `True`。这个方法把 `HttpResponse` 实例当做流式对象。

`HttpResponse.writelines(lines)`

把几行内容写入响应。不添加换行符。这个方法把 `HttpResponse` 实例当做流式对象。

F.3.4 `HttpResponse` 的子类

Django 提供了几个 `HttpResponse` 的子类，用于处理不同类型的 HTTP 响应。与 `HttpResponse` 一样，这些子类也在 `django.http` 模块中。

`HttpResponseRedirect`

构造方法的第一个参数是必须的，用于指定重定向的目标路径。可以是完全限定的 URL（如 `http://www.yahoo.com/search/`）或没有域名的绝对路径（如 `/search/`）。构造方法的其他可选参数同 `HttpResponse`。注意，HTTP 状态码是 302。

`HttpResponsePermanentRedirect`

与 `HttpResponseRedirect` 类似，不过是做永久重定向（HTTP 状态码为 301），而不是临时重定向（状态码为 302）。

`HttpResponseNotModified`

构造方法不接受任何参数，而且不能为响应添加内容。表示自用户上次访问以来页面没有改动（状态码为 304）。

`HttpResponseBadRequest`

与 `HttpResponse` 类似，不过状态码是 400。

`HttpResponseNotFound`

与 `HttpResponse` 类似，不过状态码是 404。

`HttpResponseForbidden`

与 `HttpResponse` 类似，不过状态码是 403。

`HttpResponseNotAllowed`

与 `HttpResponse` 类似，不过状态码是 405。构造方法的第一个参数是必须的，用于指定允许的方法列表（如 `['GET', 'POST']`）。

`HttpResponseGone`

与 `HttpResponse` 类似，不过状态码是 410。

`HttpResponseServerError`

与 `HttpResponse` 类似，不过状态码是 500。

如果自定义的 `HttpResponse` 子类实现了 `render` 方法，Django 认为它是在模仿 `SimpleTemplateResponse`，因此 `render` 方法必须返回一个有效的响应对象。

F.4 JsonResponse 对象

`JsonResponse(data, encoder=DjangoJSONEncoder, safe=True, **kwargs)` 是 `HttpResponse` 的一个子类，目的是辅助创建 JSON 编码的响应。它从超类继承了多数行为，不过存在一些差异：

- `Content-Type` 首部默认设为 `application/json`。
- 第一个参数 `data` 应该是一个 `dict` 实例。如果把 `safe` 参数设为 `False`（参见下文），则可以是任何可序列化成 JSON 的对象。
- `encoder` 的默认值为 `django.core.serializers.json.DjangoJSONEncoder`，用于序列化数据。
- 布尔值参数 `safe` 默认为 `True`。设为 `False` 时，可以传入任何可序列化的对象（否则只能传入 `dict` 实例）。设为 `True` 时，如果传给第一个参数的值不是 `dict` 对象，抛出 `TypeError`。

F.4.1 用法

常见的用法如下：

```
>>> from django.http import JsonResponse
>>> response = JsonResponse({'foo': 'bar'})
>>> response.content
'{"foo": "bar"}'
```

序列化字典之外的对象

若想序列化字典之外的对象，要把 `safe` 参数设为 `False`：

```
response = JsonResponse([1, 2, 3], safe=False)
```

如若不然，抛出 `TypeError`。

更换默认的 JSON 编码器

如果想使用其他 JSON 编码器类，把 encoder 参数传给构造方法：

```
response = JsonResponse(data, encoder=MyJSONEncoder)
```

F.5 StreamingHttpResponse 对象

StreamingHttpResponse 类的作用是以流的方式把 Django 的响应发给浏览器。如果生成响应耗时太长或耗费内存太多，可以这么做。例如，可用于生成大型 CSV 文件。

F.5.1 注意性能

Django 针对的是短期请求。在流式响应持续的时间段内，它将占用一个进程。这可能导致性能下降。

一般来说，应该把耗资源的任务放到请求-响应循环之外，而不是诉诸流式响应。

StreamingHttpResponse 不是 HttpResponse 的子类，因为二者 API 有点儿不同。但是，除了下述区别之外，它们的行为基本上是一致的。

- 参数为产出字符串的迭代器。产出的字符串即响应的内容。
- 除非迭代响应对象自身，否则无法访问响应内容。只能等到响应返回给客户端之后才能迭代。
- 没有 content 属性，而有 streaming_content 属性。
- 不能像类似文件的对象那样调用 tell() 或 write() 方法，否则 Django 会抛出异常。

只有十分确定把数据发给客户端之前无法遍历内容时才应该使用 StreamingHttpResponse。因为内容无法访问，所以很多中间件将失效。例如，无法为流式响应生成 ETag 和 Content-Length 首部。

F.5.2 属性

StreamingHttpResponse 对象有下述属性：

- *.streaming_content：产出字符串的迭代器，表示响应的内容。
- *.status_code：响应的 HTTP 状态码。
- *.reason_phrase：响应的 HTTP 状态描述短语。
- *.streaming：始终为 True。

F.6 FileResponse 对象

FileResponse 是 StreamingHttpResponse 的子类，为二进制文件做了优化。它使用 wsgi 服务器的 wsgi.file_wrapper（如果提供了），或者以流的形式发送文件片段。

FileResponse 期望文件以二进制模式打开，如下所示：

```
>>> from django.http import FileResponse
>>> response = FileResponse(open('myfile.png', 'rb'))
```

F.7 错误视图

为了处理 HTTP 错误，Django 自带了几个视图。使用自定义的视图覆盖这些默认视图的方法参见 F.7.5 节。

F.7.1 404（页面未找到）视图

`defaults.page_not_found(request, template_name='404.html')`

如果视图抛出 `Http404`，Django 加载一个专门用于处理 404 错误的视图。默认使用的视图是 `django.views.defaults.page_not_found()`，它生成一个简单的“Not Found”消息，或者渲染 `404.html` 模板（如果根模板目录中有）。

默认的 404 视图向模板传递一个变量，`request_path`，即致错的 URL。

关于 404 视图要注意三点：

- Django 找遍 URL 配置，如果没有找到一个匹配的正则表达式，也渲染 404 视图。
- `RequestContext` 对象会传给 404 视图，因此在视图中可以访问模板上下文处理器提供的变量（如 `MEDIA_URL`）。
- `DEBUG` 设为 `True` 时（在设置模块中），404 视图永远不会使用；此时显示的是 URL 配置和一些调试信息。

F.7.2 500（服务器错误）视图

`defaults.server_error(request, template_name='500.html')`

类似地，视图代码出现运行时错误时，Django 也会表现出特殊的行为。如果视图抛出异常，Django 默认调用 `django.views.defaults.server_error` 视图。这个视图要么生成一个简单的“Server Error”消息，要么加载并渲染 `500.html` 模板（如果根模板目录中有）。

默认的 500 视图不给 `500.html` 模板传递任何变量，而是使用空的 `Context` 渲染，以免再出现其他错误。

`DEBUG` 设为 `True` 时（在设置模块中），500 视图永远不会使用；此时显示的是调用跟踪和一些调试信息。

F.7.3 403（HTTP 禁止）视图

`defaults.permission_denied(request, template_name='403.html')`

与 404 和 500 错误一样，Django 也提供了处理 403 Forbidden 错误的视图。如果视图抛出 403 异常，Django 默认调用 `django.views.defaults.permission_denied` 视图。

这个视图加载并渲染根模板目录中的 `403.html` 模板；如果这个模板不存在，根据 RFC 2616（HTTP 1.1 规范），渲染“403 Forbidden”文本。

`django.views.defaults.permission_denied` 由 `PermissionDenied` 异常触发。在视图中如果想拒绝访问，可以这么做：

```
from django.core.exceptions import PermissionDenied

def edit(request, pk):
    if not request.user.is_staff:
        raise PermissionDenied
```

```python
    # ...
```

F.7.4 400（坏请求）视图

`defaults.bad_request(request, template_name='400.html')`

如果 Django 抛出了 `SuspiciousOperation`，可能会由 Django 的某个组件处理（例如重设会话数据）。如果没做特别处理，Django 将把当前请求视作“坏请求”，而不是服务器错误。

`django.views.defaults.bad_request` 与 `server_error` 视图非常像，但返回的状态码是 400，表示错误是由客户端操作导致的。

`bad_request` 视图也只在 `DEBUG` 为 `False` 时使用。

F.7.5 自定义错误视图

Django 自带的默认错误视图对多数 Web 应用程序来说已经够用了，但是如果需要定制行为，也能轻易覆盖，只需按下文所述在 URL 配置中指定处理程序（在别处设置不起作用）。

`page_not_found()` 视图由 `handler404` 覆盖：

```python
handler404 = 'mysite.views.my_custom_page_not_found_view'
```

`server_error()` 视图由 `handler500` 覆盖：

```python
handler500 = 'mysite.views.my_custom_error_view'
```

`permission_denied()` 视图由 `handler403` 覆盖：

```python
handler403 = 'mysite.views.my_custom_permission_denied_view'
```

`bad_request()` 视图由 `handler400` 覆盖：

```python
handler400 = 'mysite.views.my_custom_bad_request_view'
```

附录 G 使用 Visual Studio 做 Django 开发

不管你在网上听到怎样的抱怨，都不得不承认 Microsoft Visual Studio（以下简称 VS）一直是功能极其强大的*集成开发环境*（Integrated Development Environment，IDE）。作为一名多平台开发者，我尝尽了各种工具，最终还是老老实实使用 VS。

过去，影响 VS 普及的最大障碍（我觉得）是：

1. 对 Microsoft 生态系统（C++、C# 和 VB）以外的语言缺少支持。
2. 全功能 IDE 成本高。Microsoft 臃肿的"免费"IDE 对专业开发而言不那么有用了。

几年前发布的 Visual Studio Community Edition，以及最近发布的 Python Tools for Visual Studio（以下简称 PTVS）明显改变了这一糟糕状况。如今，我的所有开发工作，不管用的是 Microsoft 相关的技术还是 Python 和 Django，都在 VS 中完成。

VS 的优点就不细说了，以免你觉得我是在给 Microsoft 背书。现在我假定你至少决定试一试 VS 和 PTVS。

首先，我要说明如何在你的 Windows 设备中安装 VS 和 PTVS；然后，简要介绍一下可用的 Django 和 Python 工具。

G.1 安装 Visual Studio

安装前的提醒

VS 毕竟是 Microsoft 的产品，因此无法忽略的一个事实是，安装 VS 是个艰巨的任务。为了尽量避免少出问题，请：

1. 在安装过程中关闭杀毒软件。
2. 确保网络连接顺畅。能用有线就别用无线。
3. 关闭其他占用内存和磁盘的程序，如 OneDrive 和 Dropbox。
4. 关闭所有无需打开的程序。

做好以上几点之后，访问 VS 的网站，下载免费的 Visual Studio Community Edition 2015（图 G-1）。[1]

1. 现在的最新版是 Visual Studio Community Edition 2017，如果想下载 2015 版，请访问 https://msdn.microsoft.com/zh-cn/visual-studio-community-vs.aspx。——译者注

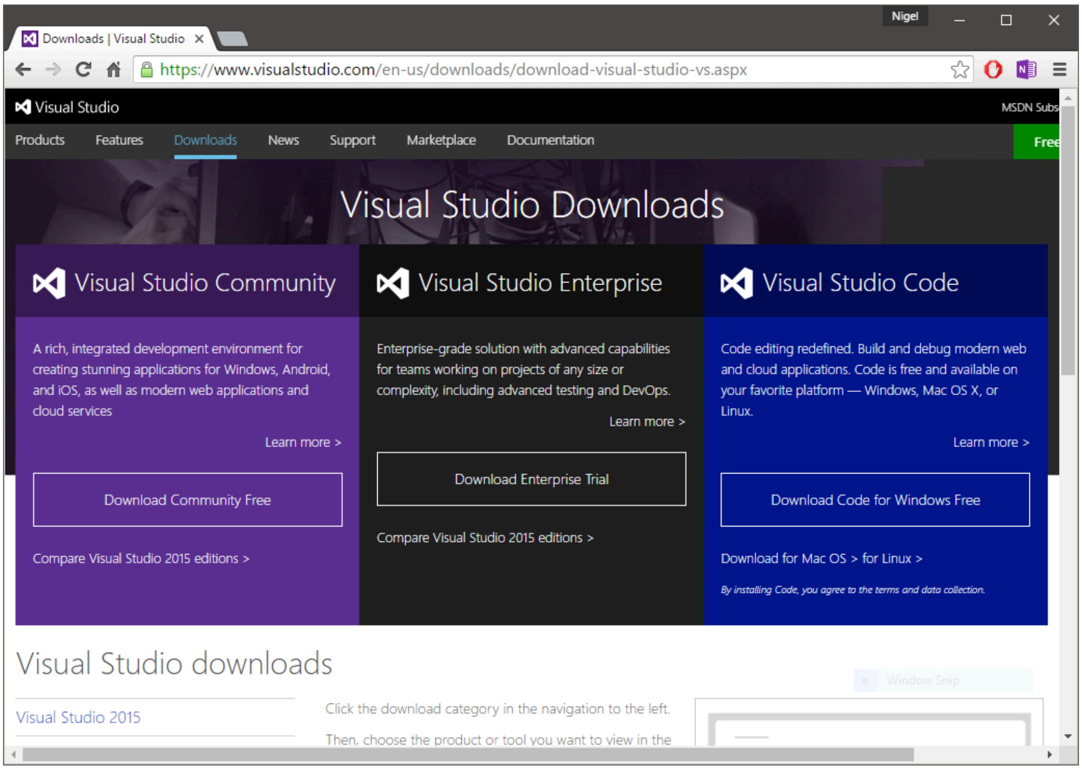


图 G-1：下载 Visual Studio

打开下载得到的安装程序文件，勾选默认的安装选项（图 G-2），然后点击"安装"。

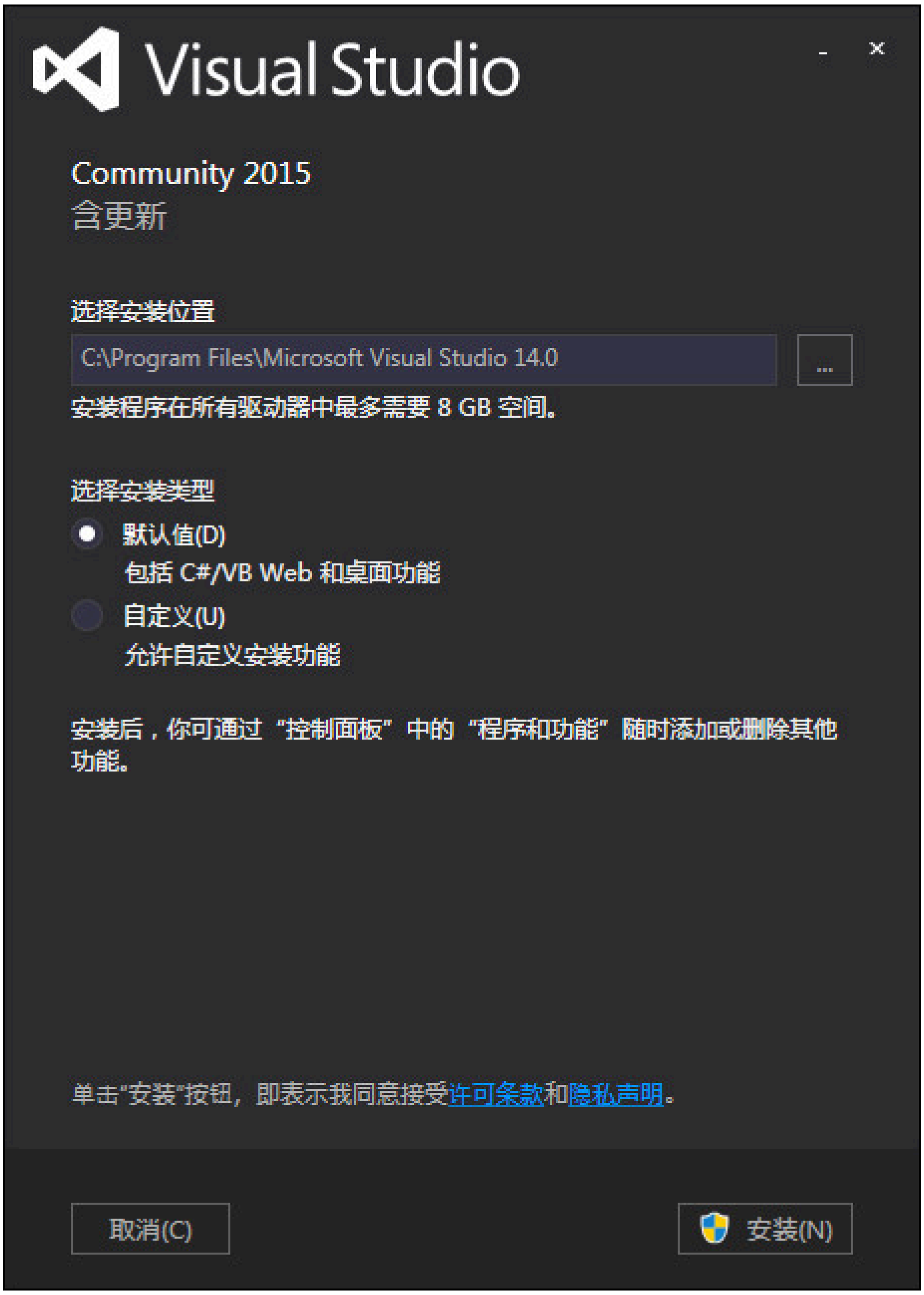


图 G-2：Visual Studio 的默认安装选项

现在，你可以去泡杯咖啡喝。可能要喝上七杯。Microsoft 的产品嘛，肯定要等很长时间。网速不同，用时也不同，可能在 15 分钟到一个多小时之间。

少数时候还可能安装失败。（根据我的经验）这基本上是因为忘记关闭杀毒软件，或者网络暂时断连了。幸好 VS 的恢复功能足够强劲，我发现安装失败后它能自动重新开始安装。VS 甚至能记住之前的位置，因此无需从头开始安装。

G.1.1 安装 PTVS 和 Web Essentials

安装好 VS 之后，添加 Python Tools for Visual Studio（PTVS）和 Visual Studio Web Essentials。

在顶部菜单中选择“工具 > 扩展和更新”（图 G-3）。

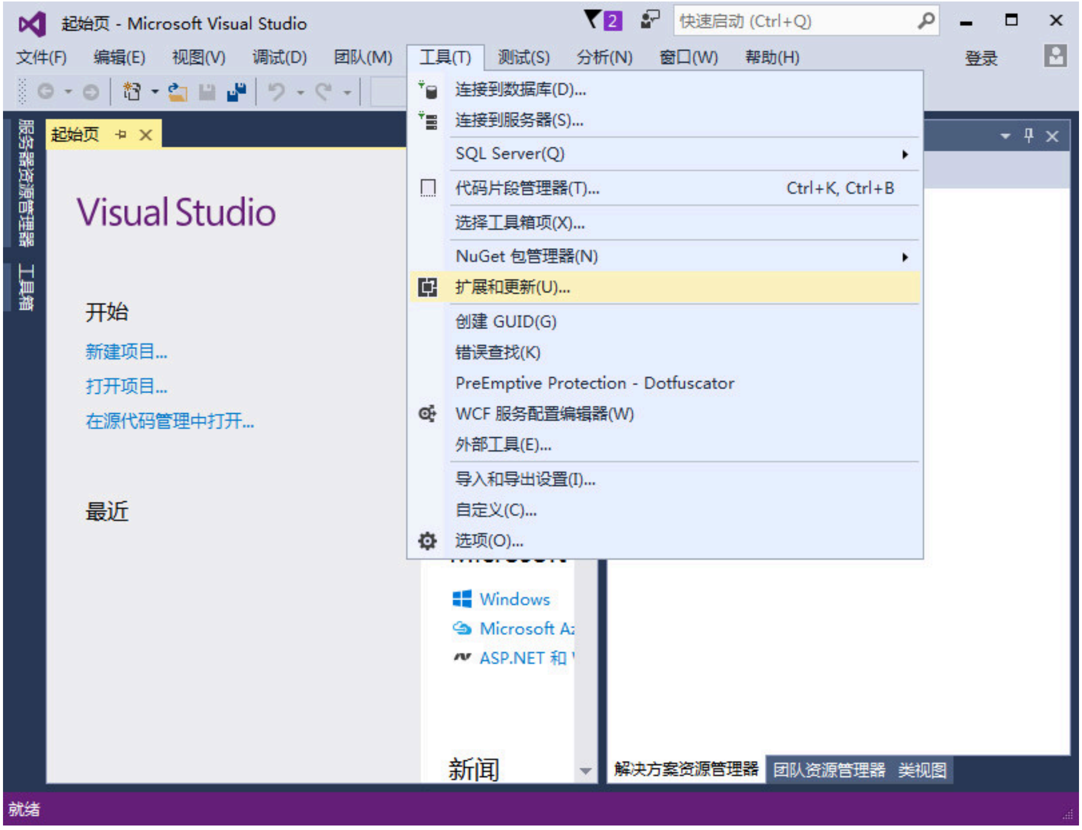


图 G-3：安装 Visual Studio 扩展

在打开的“扩展和更新”窗口中选择左边的“联机”，进入 VS 在线应用集。在右上角的搜索框中输入“Python”，第一个搜索结果应该就是 PTVS 扩展（图 G-4）。[2]

2. 如果点击“下载”按钮后打开 Visual Studio 网站，让你重新下载 Visual Studio，请跳过这一步。在 G.2.1 节新建项目时会提示你安装所需的扩展，根据提示安装即可。——译者注

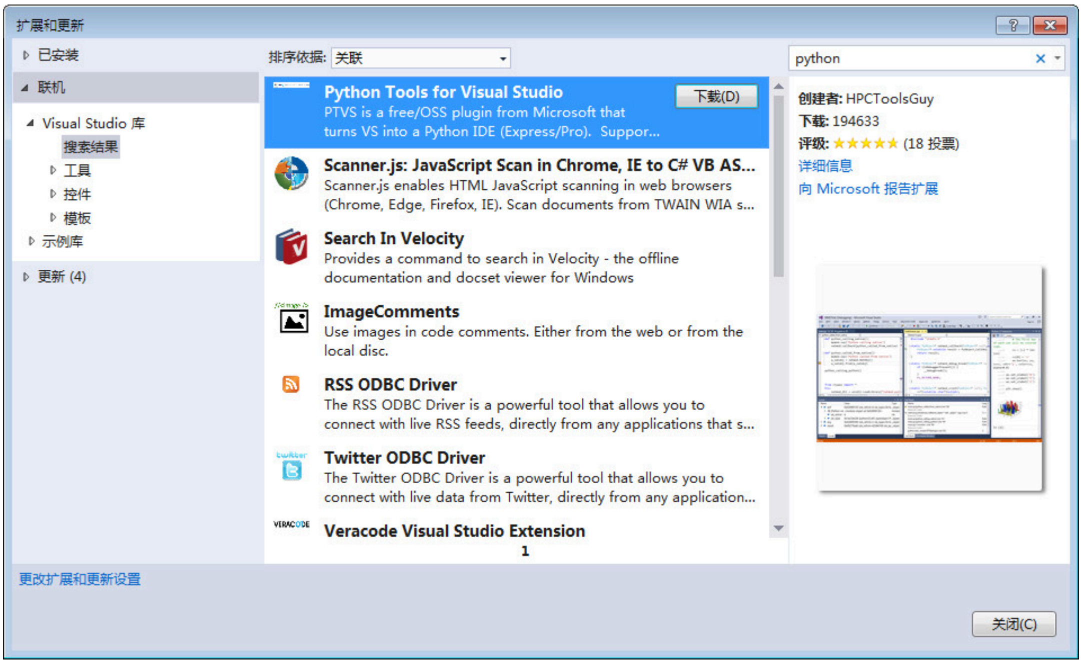


图 G-4：安装 PTVS 扩展

以同样的方式安装 VS Web Essentials（图 G-5）。注意，如果使用其他的 VS 构建版，或者之前安装过一些扩展，Web Essentials 可能已经安装了。如果是这样，“下载”按钮所在的位置会显示一个绿色对号图标。



图 G-5：安装 Web Essentials 扩展

G.2 创建 Django 项目

使用 VS 做 Django 开发最大的好处之一是，除了 VS 自身以外，只需安装 Python。如果你已经按第 1 章所述安装了 Python，就什么都不用做了，VS 会负责创建虚拟环境、安装所需的 Python 模块，甚至还在 IDE 中内

置了 Django 的所有管理命令。

为了演示这些功能，下面我们来创建第 1 章中的 `mysite` 项目，不过这一次都在 VS 中操作。

G.2.1 新建一个 Django 项目

在顶部菜单中选择"文件 > 新建 > 项目"，然后在左边的下拉菜单中选择 Python Web 项目。你应该看到如图 G-6 所示的界面。选择"Blank Django Web Project"，输入项目名称，然后点击"确定"。

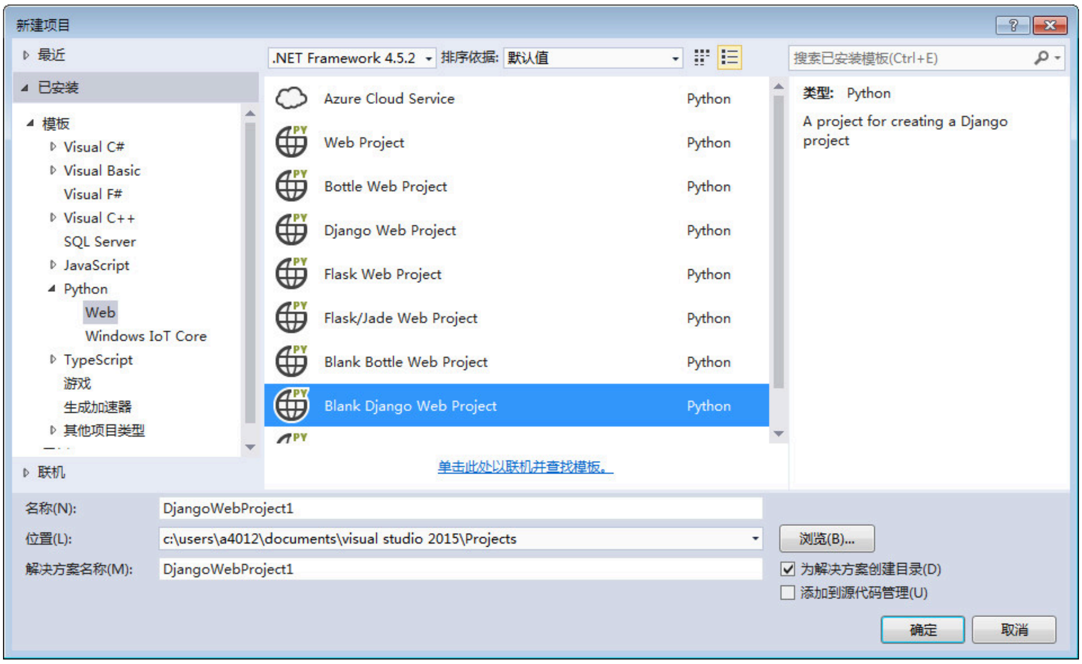


图 G-6：创建一个空 Django 项目

然后 Visual Studio 会弹出一个窗口，提示项目需要额外的包（图 G-7）。这里，最简单的选择是直接安装到虚拟环境中（选项 1），但是这样会安装最新版 Django（写作本书时是 1.9.7）。因为本书是针对 1.8 LTS 版本的，所以我们要选择选项 3，"I will install them myself"（稍后我自己安装），对 `requirements.txt` 文件做必要的改动。

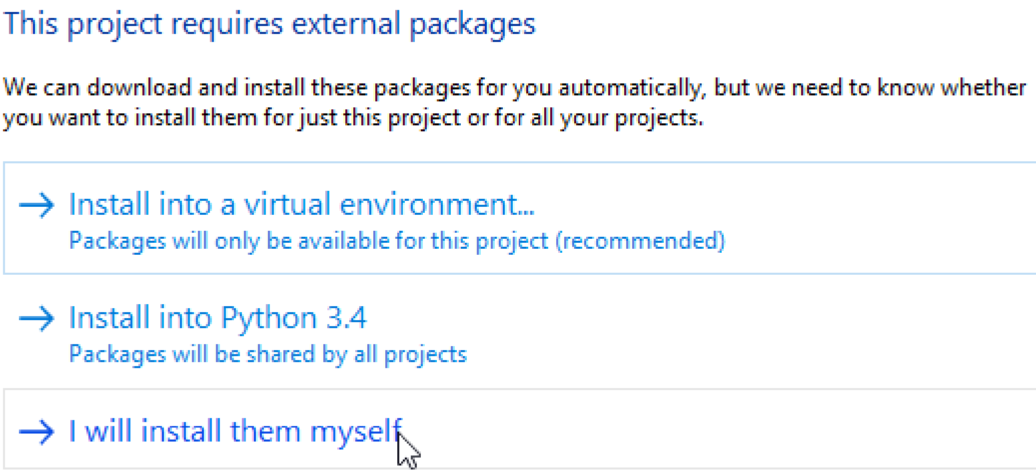


图 G-7：安装额外的包

项目创建好之后，在 VS 界面右侧的"解决方案资源管理器"中你会看到完整的 Django 项目结构已经为你创建好了。接下来要添加一个运行 Django 1.8 的虚拟环境。写作本书时，最新版是 1.8.13，因此我们要把 `requirements.txt` 文件的第一行改成：

```
django==1.8.13
```

保存文件，然后在"解决方案资源管理器"中右键点击"Python Environments"（Python 环境），选择"Add Virtual Environment"（添加虚拟环境）（图 G-8）。

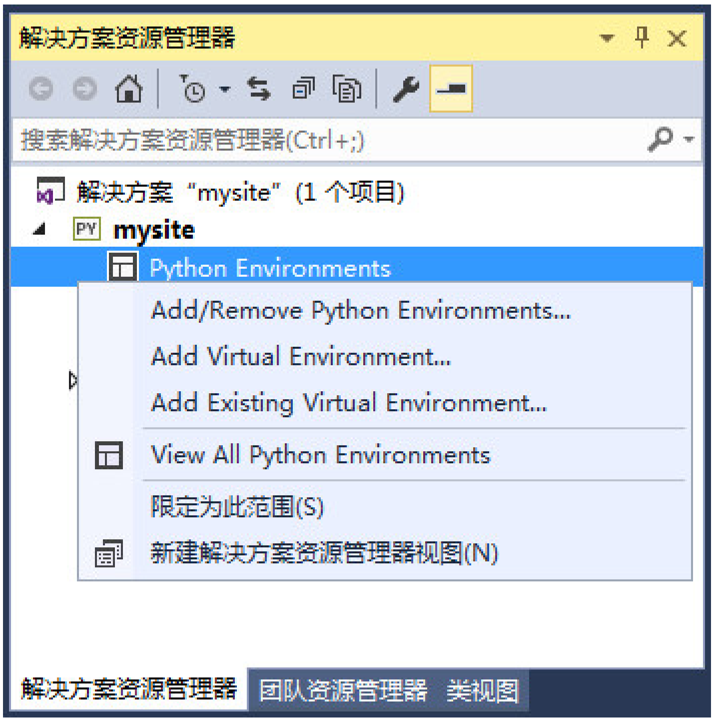


图 G-8：添加虚拟环境

在弹出的窗口把默认的环境名称"env"改为更有意义的名称（如果你跟着第 1 章的示例做，命名为 `env_mysite`）。点击"Create"（创建），此时 VS 会创建一个虚拟环境（图 G-9）。

> **提示**
>
> 使用 VS 时无需手动激活虚拟环境，因为所有代码都自动在“解决方案资源管理器”中激活的虚拟环境里运行。这样便于使用 Python 2.7 和 3.4 测试代码——只需右键点击，激活需要的环境。

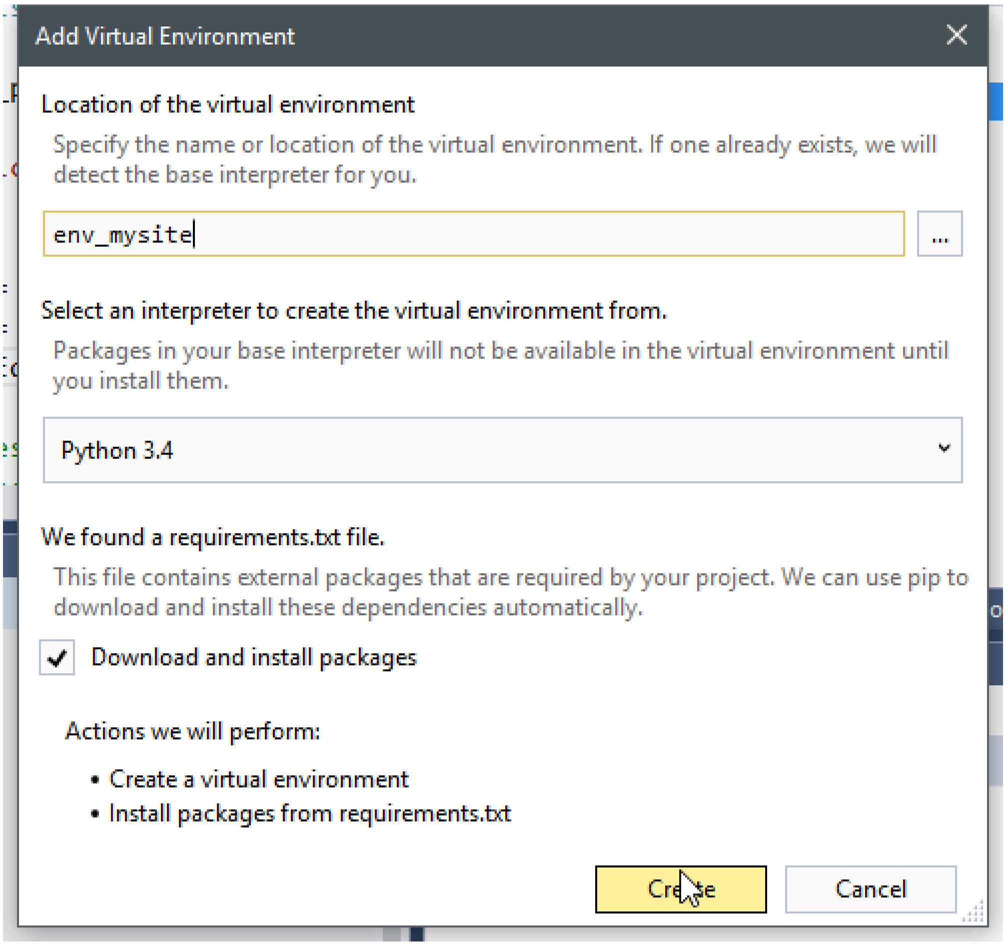


图 G-9：创建虚拟环境

G.3 在 Visual Studio 中做 Django 开发

Microsoft 做了很大努力，确保在 VS 中开发 Python 应用程序尽量简单和无痛。对新手程序员来说，最棒的功能是所有 Python 和 Django 模块都有完整的 IntelliSense 支持。这一功能能加快学习过程，无需翻阅文档，查看模块的实现。

VS 还在其他方面简化了 Python/Django 编程：

1. 集成 Django 管理命令
2. 便于安装 Python 包
3. 便于安装新的 Django 应用

G.3.1 对 Django 管理命令的集成

Django 常用的管理命令都在“项目”菜单中（图 G-10）。

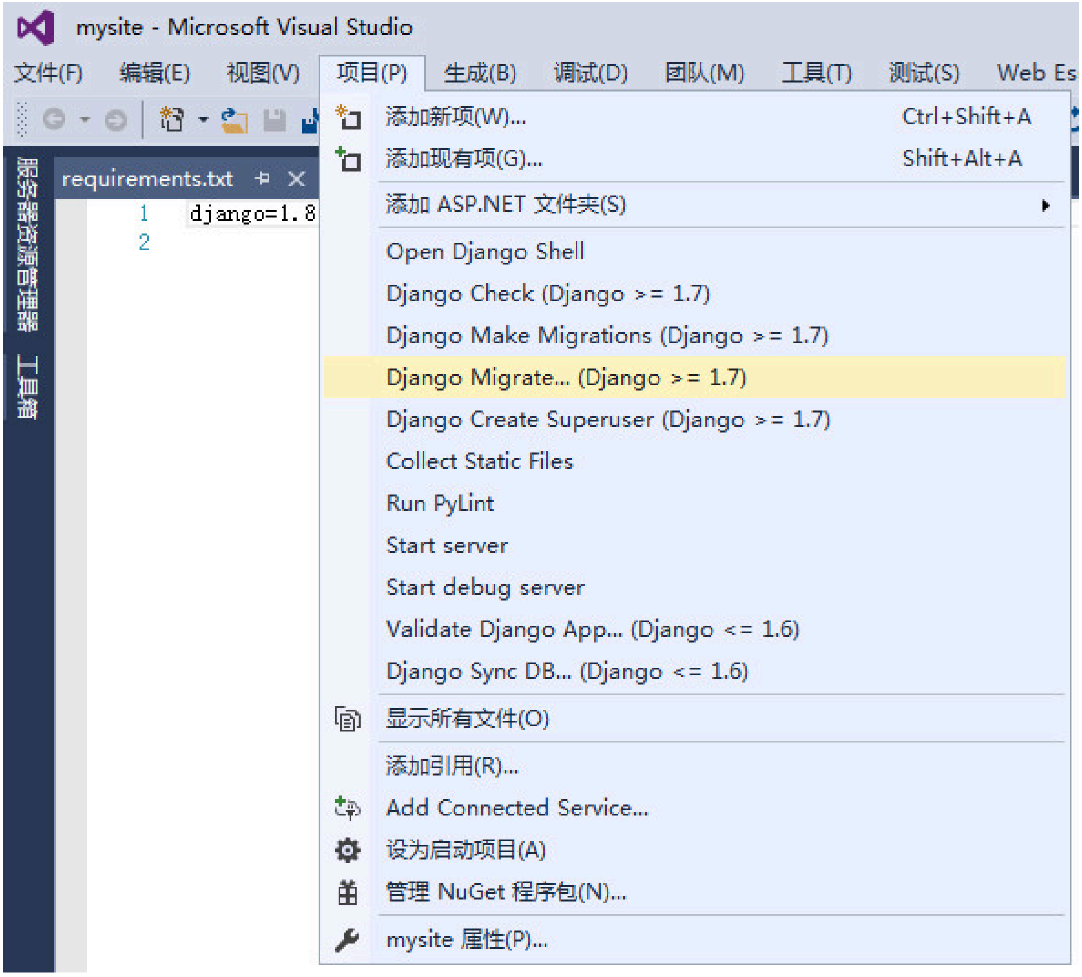


图 G-10：“项目”菜单中常用的 Django 命令

在这个菜单中可以运行迁移、创建超级用户、打开 Django shell 和运行开发服务器。

G.3.2 简化 Python 包的安装

在“解决方案资源管理器”中可以直接把 Python 包安装到任何虚拟环境中，只需在环境上点击右键，然后选择“Install Python Package…”（安装 Python 包）（图 G-11）。

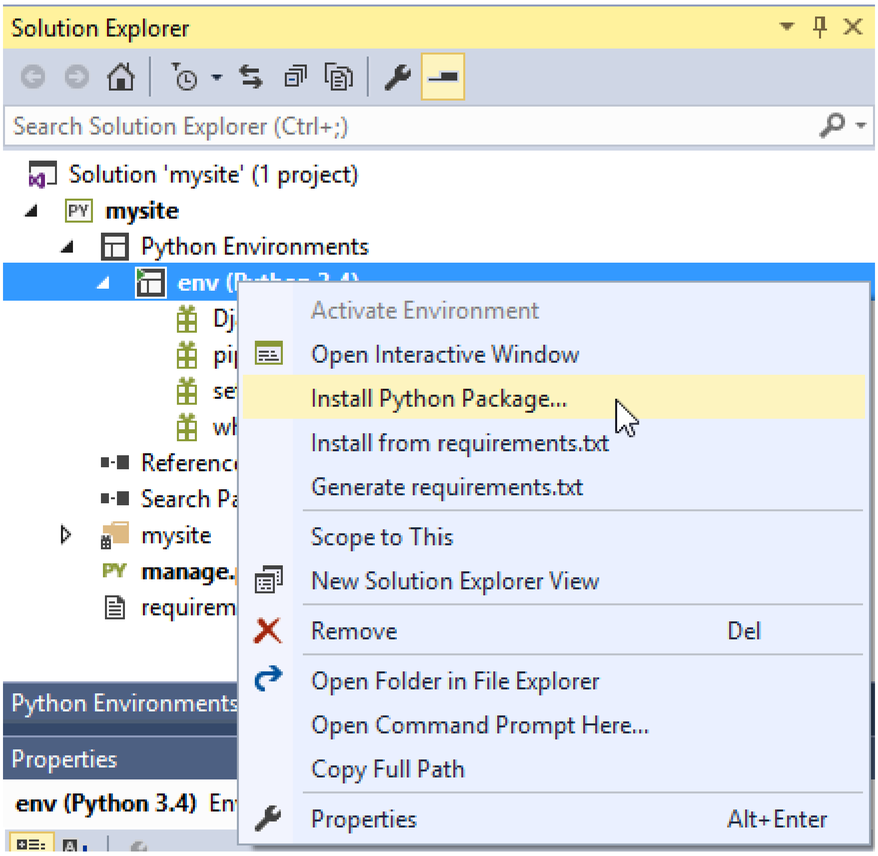


图 G-11：安装 Python 包

包可以使用 `pip` 或 `easy_install` 安装。

G.3.3 简化新 Django 应用的安装

最后，在项目中添加新的 Django 应用也很简单，只需在项目上点击右键，然后选择“添加 > Django app…”（图 G-12）。输入应用的名称，然后点击“OK”，VS 就会在项目中添加一个新应用。

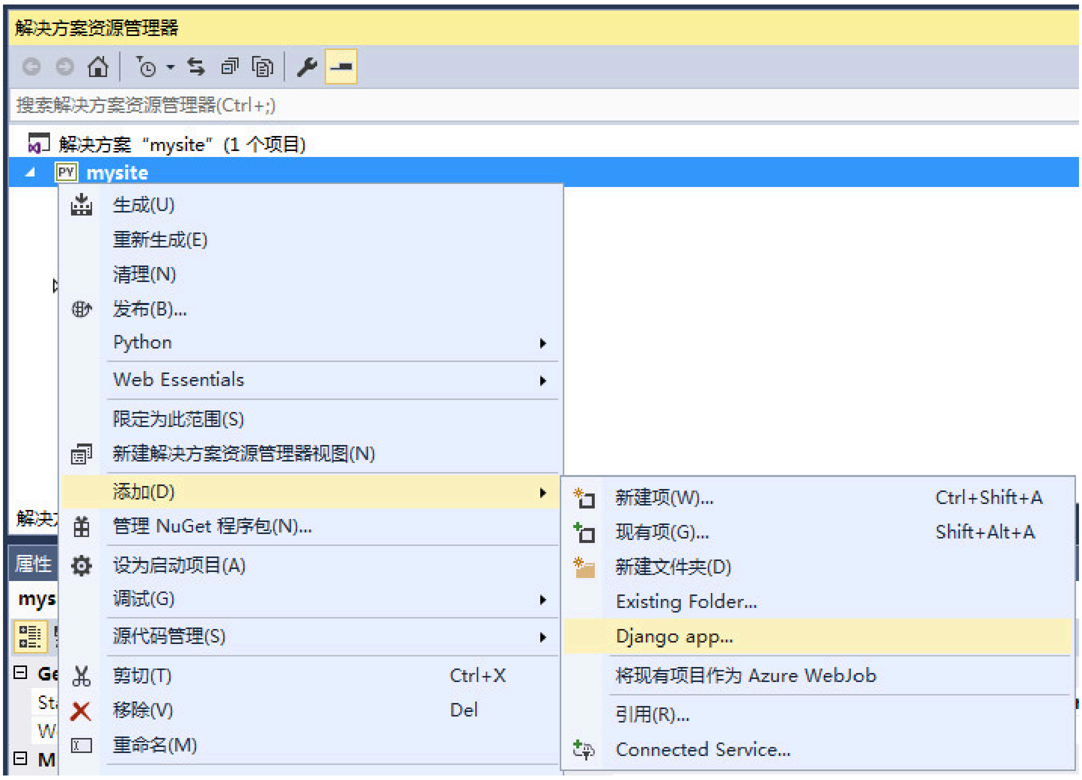


图 G-12：添加 Django 应用

以上只是快速浏览一下 Visual Studio 的功能，以便让你快速上手。还有一些功能值得一用：

- VS 的仓库管理功能，例如与本地 Git 仓库和 GitHub 深度集成。
- 使用免费的 MSDN 开发者账号部署到 Azure（写作本书时只支持 MySQL 和 SQLite）。
- 内置混合模式调试器。例如，可以在同一个调试器中调试 Django 和 JavaScript。
- 内置对测试的支持。
- 前文提过的全面 IntelliSense 支持。